UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Named Data Networking with Programmable Switches

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitetura, Sistemas e Redes de Computadores

Rui Miguel Carvalho Marques

Dissertação orientada por:
Prof. Doutor Fernando Manuel Valente Ramos

2017

Acknowledgments

To professor Fernando Ramos, for proposing so interesting a subject for my dissertation, and supervising it.

To my open office coworkers for the friendly, fun mood, the evening karaokes and all the giggles.

To Mihai Budiu of VMware, Antonin Bas of Barefoot Networks and Nate Foster of Cornell University for the continued attention to my e-mail messages and my interventions on the p4lang GitHub repositories.

To Engineer Jeferson Santiago da Silva, MSc by Polytechnique Montréal, for responding to my e-mails regarding P4 externs so promptly and kindly.

To Salvatore Signorello, for reviewing my written draft and pointing errors in it.

To my (some current, some former) colleagues Ana Pereira, João Ferreira, Henrique Califórnia Mendes and Inês Gouveia for approaching me on my first spooky week in the Faculty of Sciences and integrating me into their group. If it hadn't been for that, I might have given up on university.

To former colleagues Ricardo Tomaz and (again) Ana Pereira for the good times during our first year and the endless laughters.

A special one to the conference group at my secondary school for being the most serious people I know. In particular, to Professor Armando Pedrosa.

Another special one to Professor Guilherme Arroz of IST-ULisboa and doctor Luísa Barreto for being a source of encouragement.

And a final one to my family for the support!

Funding This work was partially supported by the European Commission through project FP7 SEGRID (607109) and project H2020 SUPERCLOUD (643964), and by national funds of Fundação para a Ciência e a Tecnologia (FCT) through project UID/-CEC/00408/2013 (LaSIGE).

Resumo

As redes IP, que são universais atualmente, apresentam um conjunto de problemas que encontra a sua génese nos seus propósitos originais. Na génese do IP, o objetivo era a partilha de recursos. Hoje em dia, as redes de computadores já não se baseiam num computador *mainframe* a disponibilizar recursos de hardware. São usadas como meio de disseminação alargada de uma panóplia de média, como ficheiros de hipertexto, imagens e vídeos.

Grande parte das dificuldades no uso das redes IP advém do uso de endereços. A necessidade do endereço como identificador indispensável à comunicação obrigou ao aparecimento de esquemas complexos à medida que as redes cresceram: refere-se o DNS, o DHCP e a gestão de prefixos associada às unidades autónomas (*autonomous systems*, conhecidas também pela sigla AS). Já quase se esgotou o espaço público de endereços, para além de que a gestão dos reservatórios públicos e privados é um processo complicado e propenso a erros.

A par deste panorama, o hardware da rede foi otimizado para desempenho e os dispositivos de encaminhamento (*routers*) e comutação (*switches*) tornaram-se caixas negras fechadas, correndo vários protocolos em hardware para maximizar o desempenho. O software dos hospedeiros (*hosts*), por seu turno, materializou-se na API de sockets que usamos até hoje.

As redes baseadas em nomes (*named data networks*, ou NDN) divergem fundamentalmente da rede IP. Enquanto que a última tem como objectivo transportar um pacote para um destino com base no seu endereço, as NDN não fazem qualquer uso de endereços. O problema é reformulado em como levar dados com um determinado nome de um produtor para os consumidores. Assim, na rede NDN circulam apenas dois tipos de pacotes: ***Interest*** e ***Data***. Os pacotes *Interest* são emitidos por consumidores que procuram dados. Estes dados estão globalmente e univocamente ligados a um pedaço de informação. A rede NDN trata de encaminhar o pedido Interest até um produtor. Em resposta, o produtor emite um pacote *Data*, que alberga, no seu interior, o pedaço de informação correspondente ao que foi pedido no Interest.

Os nomes são o centro da NDN. Podem ser *flat*, mas também podem ser utilizados de forma hierárquica. Por exemplo, "ulisboa/fciencias/index.html"é um nome hierárquico.

Cada troço do nome separado por '/' chama-se um **componente**. Esta hierarquia é fundamental para conferir contexto ao nome e escalar a NDN.

Esta mudança de paradigma oferece vantagens. Desde logo, o endereço torna-se desnecessário, evitando assim processos de gestão intermédios, exaustão do reservatório de endereços públicos e o uso dos *Network Address Translators* (NATs). Para além disso, todos os pacotes Data vêm assinados, pelo que as NDN dispõem de segurança inerente e, a par disto, de uma maior dificuldade em atacar alvos específicos, dado que todos os dispositivos na rede, quer nós intermediários quer hospedeiros, estão desprovidos de identificação. Os problemas de segurança existem, mas reduzem-se, desta forma, a distribuição de chaves segura, buracos negros (*black holes*) e ataques *distributed denial of service* (DDoS).

O encaminhamento em NDN é semelhante ao que ocorre no IP, mas, em vez de manter endereços de 32 bits nas tabelas de comutação, os encaminhadores utilizam os nomes, de comprimento arbitrário para decidir por onde encaminhar os pacotes. As tabelas são populadas de modo análogo ao que acontece no IP, por exemplo através de um protocolo *link-state* para NDN, homólogo do OSPF. As tabelas de comutação têm, portanto, pares *(string, integer)*, associando um nome a uma dada interface (que pode ser física ou lógica) do dispositivo. Quando um nome faz *match* com mais de uma entrada, é selecionada aquela que tiver maior número de componentes (e, portanto, for o prefixo mais comprido).

Quando recebe um pacote Interest, o encaminhador consulta a sua *content store* para verificar se pode servir o conteúdo diretamente. Nesse sentido, a content store é uma funcionalidade absolutamente fundamental neste paradigma, pois permite trazer o conteúdo para perto dos consumidores. Este caching feito ao nível da rede é considerado uma das grandes mais-valias das NDN. Se puder servir diretamente, fá-lo. Caso contrário, o Interest segue para a Tabela de *Interests* Pendentes (*Pending Interest Table*, ou PIT), onde se mantém registo dos pedidos na forma de uma lista de interfaces que pediram um determinado nome.

Se esta lista está vazia, então este Interest é o primeiro pedido para este nome. Será assim encaminhado para uma interface do dispositivo determinada pela FIB (*forwarding information base*). A FIB mantém associações *(nome,interface)* e comuta o pacote se encontrar um nome que seja prefixo do nome inscrito no Interest em processamento; realiza, deste modo, um *longest prefix matching* de nomes à granularidade do componente. Se a lista não está vazia, então outra interface já requisitou os mesmos dados. Nesse caso, o encaminhador pode descartar o Interest que acabou de receber, pois o pedido para esse nome já foi anteriormente lançado. Apenas adiciona à lista a interface de onde veio este Interest repetido.

Quando recebe um pacote Data, o encaminhador consulta a PIT para verificar se está à espera de dados para este nome. Se não, então descarta o pacote. Se está, efetua uma difusão *multicast* para todas as interfaces que registou na lista para o respetivo nome.

Desta forma se consegue garantir que os conteúdos requisitados chegam a todos os consumidores que os pediram. Se os campos de metadados do Data assim o permitirem, o encaminhador também armazenará o Data na *content store*.

Um dos grandes problemas deste novo paradigma é a sua materialização prática. Como vimos, um encaminhador NDN é fundamentalmente diferente do seu homólogo em IP, ou de qualquer outro tipo de comutador de pacotes, e por isso não é possível adaptar equipamento tradicional para NDN. Recentemente, porém, foram propostos comutadores programáveis, alguns já em produção (e.g., Tofino da Barefoot Networks). Estes dispositivos permitem definir precisamente o modo como o equipamento de rede processa pacotes, e reprogramá-lo sempre que necessário.

Todavia, programar estes dispositivos, utilizando a sua interface de baixo nível, é quase como programar em microcódigo, e portanto não se trata de uma tarefa fácil. Esta lacuna foi uma das motivações para a linguagem de alto-nível, P4.

A linguagem P4 surge no seio das redes programáveis, bem como das redes definidas por software, propiciada pela rigidez do OpenFlow quanto ao conjunto de protocolos que suporta. No entender dos seus criadores, um OpenFlow ideal seria aquele que permitisse ao operador de rede definir os seus próprios cabeçalhos e criar os seus próprios protocolos. Assim, a linguagem P4 tem três grandes objetivos. Primeiro, não estar dependente de nenhum protocolo específico, permitindo, pelo contrário, que estes sejam definidos pelo controlador. Segundo, poder ser reconfigurada pelo plano de controlo a qualquer momento. Terceiro, não estar dependente de nenhuma arquitetura subjacente; isto é, poder ser escrita (e depois compilada) para um qualquer dispositivo da mesma maneira que um programa escrito na linguagem C pode ser escrito sem preocupações relativamente à arquitetura de processador subjacente.

O consórcio P4 oferece um compilador *front-end* que transforma a linguagem P4 numa representação intermédia (IR), enquanto que o vendedor do dispositivo disponibiliza um compilador *back-end* que processa a IR e traduz para a linguagem própria do aparelho. Um dispositivo assim concebido é **compatível com P4** (*P4-compatible*). Com P4, é assim possível definir cabeçalhos, *parsers* e uma sequência de tabelas de *match-action* à escolha para um qualquer aparelho compatível e definir as ações a partir de um conjunto de primitivas oferecidas pela linguagem.

Neste trabalho, propomos a conceção de um *router* NDN utilizando a linguagem P4. O nosso trabalho parte de um anterior chamado NDN.p4 por Signorello *et al*, que foi implementado na versão 14 do P4 (abreviado P4-14). É um protótipo do encaminhador NDN, proporcionando uma tabela de interesses pendentes (PIT) e uma FIB. A PIT é descrita utilizando registos, que mantêm estado no comutador P4. A capacidade de ter estado e de o manusear é indispensável para implementar um encaminhador NDN, como se deduz da nossa anterior descrição sobre o respetivo funcionamento.

O trabalho anterior tem, no entanto, limitações. Em primeiro lugar, não dispõe de

uma *content store*, uma das peças principais deste paradigma. Para além disso, o recurso a matching ternário e exato tem problemas de escalabilidade e não há suporte a multicast de pacotes Data.

Nesta dissertação, desenhámos e construímos, da forma mais modular e genérica possível, um encaminhador NDN utilizando a versão mais recente do P4, P4-16, providenciando, para além da PIT e da FIB — que faz longest prefix matching utilizando um método inovador —, uma *content store* implementada quer em registos, quer diretamente no *switch* P4. As principais inovações são as seguintes:

- Implementação da *content store*, que armazena pacotes e permite ao encaminhador NDN servir pedidos de imediato. Numa primeira versão, concebêmo-la em registos P4. A segunda versão é editada diretamente num *target*, o *simple_switch*.
- Utilização de um método inovador para conseguir *longest prefix matching* (lpm) em redes NDN. O método proposto por Signorello *et al* apoia-se num mecanismo relativamente complicado de *matchings* ternário e exato que não escala bem. Optámos por utilizar diretamente o método lpm, mantendo a ideia de efetuar *hashing* dos componentes do nome. Os resultados dos cálculos de dispersão dos componentes são concatenados pela mesma ordem em que aparecem os componentes respectivos. O produto final desta concatenação figura assim como entrada na tabela, com uma máscara de rede que será o comprimento do resultado da função vezes o número de componentes.
- Realização de *multicast*, quer em linguagem P4 (neste caso, para um número máximo de portos, devido à falta de mecanismos de iteração), quer diretamente no *software switch*.

Finalmente, avaliámos a nossa solução através de vários testes de funcionalidade e comparámos ao NDN.p4 em termos de espaço ocupado pelas entradas da FIB.

Palavras-chave: redes programáveis, nomes, P4, encaminhadores

Abstract

Named data networks (NDN) differ substantially from traditional TCP/IP networks. Whereas the TCP/IP communications stack focuses on delivering a packet to a destination based on its address, NDN abolishes the use of addresses and reformulates the problem as how to fetch data with a given name and bring it closer to its consumers. For this purpose, consumers emit Interest packets, writing the name for the resource they desire. The network routes that packet to a producer of the data uniquely associated to that name. NDNs achieve this by employing routers whose functions are similar to those of traditional networks, with a central difference: they route Interests based not on an address, but on a name. Another fundamental innovation of this paradigm is the introduction of a content store in NDN routers. This gives the ability to perform caching in the network, and as such is key to improve network efficiency.

The main challenge of NDN is that of deployment. As the design is radically different, current routers and switches cannot be "extended" to offer NDN. Fortunately, the emergence of programmable switches, and of a high-language level to program them (such as P4), gives hope for the state of affairs to change. With P4 it is possible to define precisely how packets are processed in these programmable switches, allowing the definition of headers, parsers, match-action tables, and the entire control flow of packets in the switch.

In this dissertation we propose the design of an NDN router and implement it using the P4 language. We improve over previous work in two main aspects: our solution includes, for the first time, a content store. In addition, we propose an innovative method to perform longest prefix matching that requires significantly less memory per route than the former, allowing the FIB to scale more easily. We evaluate our solution using P4 switches, in terms of the main NDN functionality required.

Keywords: P4, NDN, routers, programmable, switches

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The primary use case of the Internet has changed, from a simple connection-oriented message passing paradigm to distribution of a large plethora of diverse media, such as hypertext, images and videos. As a result, Internet system developers struggle to meet their requirements when distribution comes into play.

One of the solutions for this problem is to change the focus of the network from connections to data. Named data networks (NDN) [32] are a new architecture that aims to jointly solve many of the problems of IP as a whole by following this approach.

NDN switching equipment routes content based on naming data, instead of computer addresses. The routers cache data whenever possible (in a structure called the **Content Store**), multicast it to its consumers, and suppress many of the complex mechanisms to map data to locations, such as the Domain Name System (DNS). In addition, NDNs have some inherent security design, by authenticating content and making targeted attacks more difficult.

In short, the named data network is an architecture that much better fits today's Internet than the IP network. This better conformity stems from NDN directing its focus to what the Internet is most commonly trying to achieve: distributing content.

1.1 Motivation

There is a substantial difference between the requirements of an NDN router to those of an IP router. As a consequence, despite the fact that the concept was proposed over a decade ago, no production hardware has ever been built specifically for NDN. Caching data in a network forwarding device is indeed atypical. Software constructs, such as NFD [16], are useful for experimentation with current and novel features of NDN technology, but are limited in terms of performance, and the solutions cannot be easily exported to switching hardware.

With the advent of programmable networks [22], building production NDN systems may now be closer to becoming a reality. Because network devices can now be pro-

grammed [18], it becomes easier to make forwarding devices run new protocols. These capabilities gave birth to novel programming languages to express a wide range of functionality, putting a premium on expressiveness and portability, such as P4 [17], the first language to program switches.

On this regard, NDN.p4 [26] was an important step. This was the first attempt to implement an NDN router using the P4 language. This has two great advantages. First, it enables NDN in real switches, as P4 is designed to be compiled down to forwarding hardware. Second, this language has attracted the attention of several renowned companies and its adoption is growing rapidly [2], so it is expected to become the norm to program switching hardware. In fact, Barefoot Tofino, the first programmable hardware switch, already supports P4.

However, NDN.p4 still bears a number of limitations. Namely, it does not provide a Content Store, which is the main key to NDN's success as a distribution network. Second, its FIB matching process does not to scale well. Finally, it does not distribute content correctly when multiple clients request the same content (i.e., it does not offer multicast).

Furthermore, NDN.p4 was implemented using version 14 of P4 (P4-14 for short). This version is expected to be deprecated soon. The new version of P4, P4-16, provides a stable programming language definition, that promotes backwards-compatibility. In addition, it gives more freedom for switch designers to include novel data plane functionality outside the scope of P4, but that can be used by P4 programs via an extern primitive. The fact that this will most certainly be the main version of the language used in production systems, motivates us to focus our attention in P4-16.

1.2 Objectives

Our goal in this dissertation is to implement an NDN router in P4's most recent version, P4-16, that goes beyond that of NDN.p4 and bridges its constraints, by handling Data packets and multicast, by caching content in the Content Store, and by improving the FIB scalability.

1.3 Contributions

The main contributions of this dissertation are the following:

- The design and implementation of an NDN router using the latest version of P4, P4-16, for the first time;
- Our NDN solution improves the previous [26], by including a content store, an innovative method for longest prefix matching, and the ability to multicast Data packets;

- We evaluate the functionality provided to demonstrate its correctness and the conformity of our implementation to NDN's core protocol.

In the midst of our progress, we contributed to the P4 language and community by reporting a number of compiler bugs [6][7][14][15], helping others [9][10] and suggesting improvements [13].

The result of this work, including the tools developed for testing purposes as reported in §5.1, are available publicly in our working group's repository[1].

1.4 Document structure

The document is structured as follows. We first study the background and related work in §2, including NDN.p4. Then, we look into the design of our new NDN router using P4, in §3. Following that, we explain our implementation in §4. We finish with an evaluation composed of several feature tests to prove the correctness of our implementation, in §5, and conclude in §6.

[1] https://github.com/netx-ulx/NDN.p4-16

Chapter 2

Background & Related work

The purpose of a computer network, such as the Internet, at the time of its genesis in the 60's and 70's, was to let computers share resources. The resulting communication model was basically a conversation between two hosts: one wishing to use a resource, the other willing to share it. This motivated the use of (source and destination) addresses [21], both to identify *and* locate a host in order to achieve communication.

Many of the problems that the Internet faces today are due to that ancillary, simplistic purpose, on top of the forecast that the network would never be larger than a few thousand hosts. Computers then became unexpensive commodities. The network grew and proliferated. Now widespread, it faces several difficulties, including:

- The use of addresses and the incompatibility of the Internet's original design with the use cases of the modern Internet. Availability[1], security and location-dependence issues surface, leading to complicated retrieval schemes like CDNs (content distribution networks) and P2P networks, as well as complex name-to-location mapping workarounds such as DHCP and DNS [21]. On top of it, the reservatory of public addresses is exhausted, forcing a change to IPv6, that is yet to be fully materialized;

- Switches and routers are closed, black boxes interpolated in the network, performing functions tailored in hardware for maximum performance, since general-purpose CPUs have remained two orders of magnitude slower than dedicated switching chips [18]. They are hard to manage and must be configured individually [22]. The set of supported protocols they run is fixed and rigid. Supporting new ones requires changes in hardware, and the hope that the manufacturer will include support in its new generation of equipment — and, even so, with results seen only several years later [25];

- The Sockets API was created around 30 years ago to tackle the very specific problem of supporting TCP in BSD Linux and providing an interface to its users. The

[1] We invite the reader to [24], which introduces a formal reflection on this topic.

Internet is now predominantly data and service oriented (as opposed to connection-based), but the sockets API, based on a connection-oriented model, has a solid grip as the most widespread communications toolbox throughout [19].

Several solutions have been proposed to resolve these problems, but changes happen slowly. In this chapter, we first present the two that promise to modify the state of affairs, and that are central to this dissertation: named data networks, in §2.1, and programmable networks, in §2.2. In §2.3 we describe the first solution — and only to the moment, to the best of our knowledge — that proposes an NDN router using programmable switches. We include the limitations of this solution that are the main motivation for our work.

Tangent related work includes good practices on design and implementation for NDN software [31]. In this paper, authors divides routers into two categories, core and edge, and proposes a relaxation of requirements for core routers. On the topics of stateful forwarding devices, [29] designs (but does not build) *Caesar*, a stateful, fast packet processing router for information-centric networks (a supergroup of which NDN is an instance).

2.1 Named Data Networks (NDN)

Named data networks (NDN) [32] follow the content-centric approach [21], differing significantly from traditional IP networks. They provide an unified way of solving the issues flagged earlier at the beginning of this chapter as a whole. Whereas IP networks, such as today's Internet, still base communications on location, Jacobson et al argue that the focus should be shifted to the information itself, by naming data and not endpoints [21]. Accordingly, the problem the network aims to solve changes from *"deliver a packet to a destination identified by this address"* to *"fetch data identified by this name"* [32].

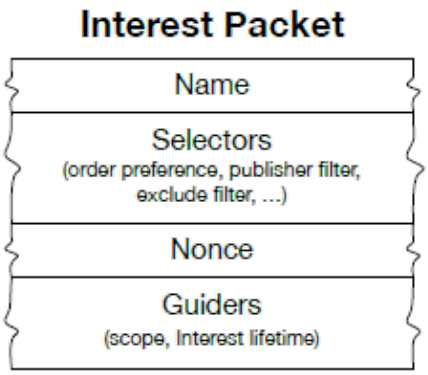

Figure 2.1: A NDN Interest packet [32].

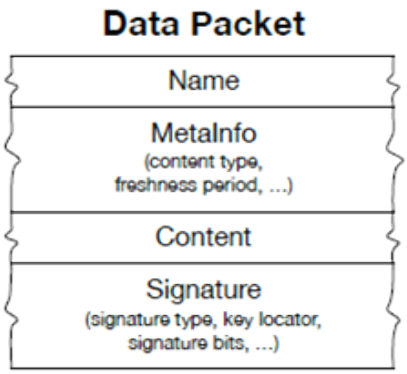

Figure 2.2: A NDN Data packet [32].

An NDN is composed of hosts and **routers**. Of these hosts, some – the **consumers** — desire named resources[2]. Others — the **producers** – offer them. Two types of pack-

[2]We call them, interchangeably, as resources, data, or assets.

ets circulate in this network: **Interest** and **Data** (see Figure 2.1 and Figure 2.2), both of which include the name of an asset. Communication is propelled by the consumer, who puts the name of the resource he desires in an Interest packet and flushes it into the network. The network will route it **upstream** to a node that contains the corresponding asset. That node, the producer, will then build and digitally sign a Data packet carrying the requested content. The signature binds the name to the content. The network then returns it **downstream** towards the consumer who emitted the request.

Names are at the heart of named data networks. They can be flat, which is suitable for local environments, but hierarchical namespaces are essential to contextualize data and scale the routing system [32]. Names can therefore be divided into **components**, which are typically separated by '/', similar to how URLs are structured (see Figure 2.3). Data that is globally accessible necessarily has a globally unique name.

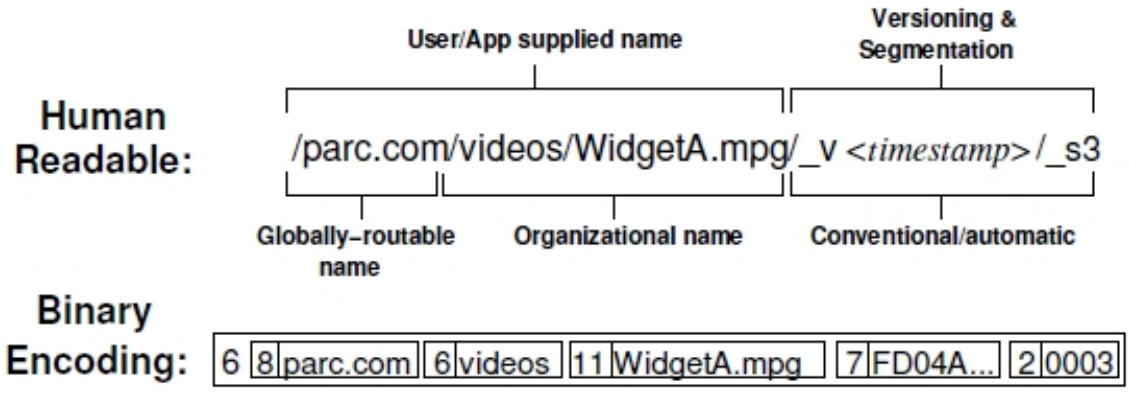

Figure 2.3: An example of an asset name [21].

We will now cover how routers operate and how packets are encoded when sent to the network.

2.1.1 Routing and forwarding

Routers perform their function by relying on names. When forwarding Interests, the router looks into its forwarding table for a matching prefix. If multiple prefixes match, then the router chooses the longest. For example, if the router contains two route entries, one for "*pt/ulisboa*" and another for "*pt/ulisboa/fciencias*", then an Interest targeting "*pt/ulisboa/fciencias/index.html*" will match the latter entry. This **longest prefix matching** with a component-based granularity is therefore similar to how IP lookups operate, but, instead of using 32-bit (or 128-bit, in the case of IPv6) integers, they use names. These FIBs are updated through routing protocols; for example, NSLR is an equivalent of the OSPF protocol for NDNs.

In order to fulfill their duties, routers are equipped with three data structures (see Figure 2.4).

- The **Pending Interest Table (PIT)** maps names to a list of interfaces[3]. When the router receives an Interest, it adds the interface from where it arrived to the list for that name.
- The **Forwarding Information Base (FIB)** maps names to outgoing ports. When multiple nodes request the same resource, the router forwards only the first Interest upstream. The remaining ones are only recorded on the PIT;
- The **Content Store** caches Data packets temporarily. If the producer permits it (by filling certain fields on the Data packet), the router can archive it and serve directly whenever it finds an Interest for the same name.

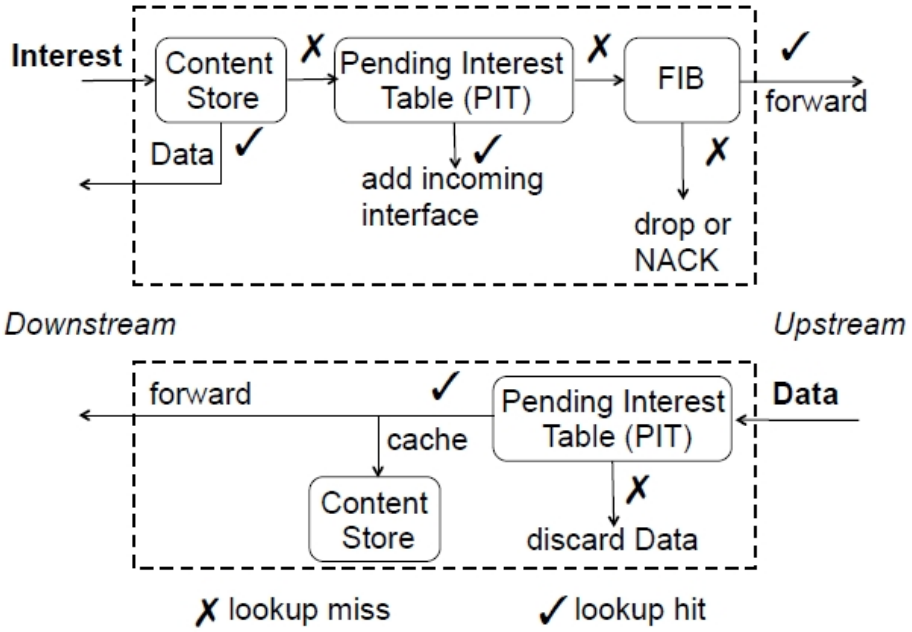

Figure 2.4: Forwarding on an NDN router [32].

This is the essential setup for the NDN router. Some literature [30][32] proposes extra functional blocks, such as an adaptive forwarding module and a FIB with multiple outgoing ports for the same name, for fault tolerance.

The two main advantages of NDNs are the possibility of in-network caching, that brings information closer to its users, and its inherent security. With respect to the latter, devices have no identification. Hence, directing attacks at specific targets becomes much harder. At most, one could try to forge bogus Data packets, but, since they are signed, the consumer will discover the integrity of the packet has been compromised when verifying the signature. The set of security problems is thus mostly reduced to distributed denial of services, black holes, and key distribution.

[3] In this text we use the terms "interfaces" and "ports" interchangeably. Nama data networking literature sometimes uses the word "face" as a hyperonym for both.

2.1.2 Packet encoding

At the network level, NDN packets are structured as a hierarchy of Type-Length-Values (TLVs) [4]. A TLV is a structure that contains, in this order:

- An unsigned byte named **type**;
- An unsigned byte named **length** (we also call it "**len**" or "**lencode**");
- And a string of arbitrary octets known as **value**.

The type is up for applications using TLVs to interpret. Length expresses the size of the Value that follows. The maximum size is 252 bytes; if the lencode has a value of 253, 254 or 255, then there is an extra field sitting after the lencode. We henceforth call it "***extended length***" or "***length extension***". It expresses the real size of the Value that follows (see Figure 2.5).

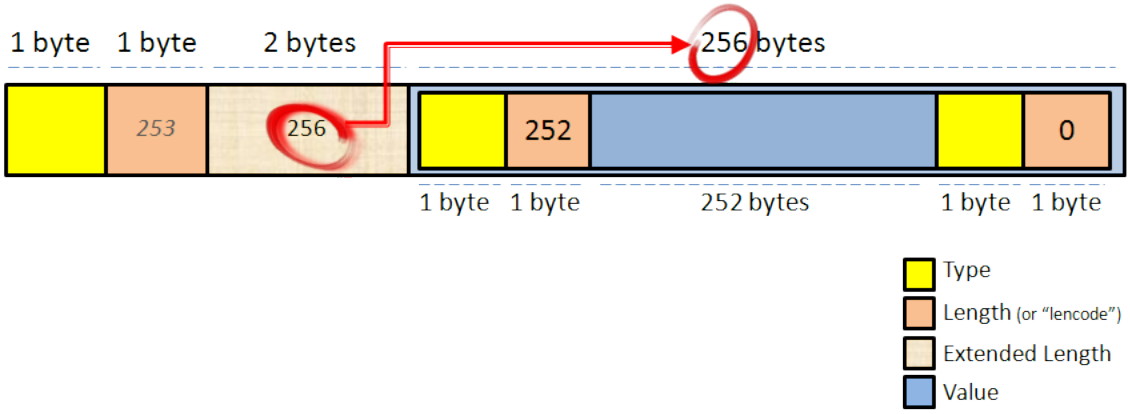

Figure 2.5: An example of a Type-Length-Value.

- If lencode has a value of 253, then the length extension is 2 bytes, and the dimension of the TLV value can go up to 2^{16} bytes[4] (64 megabytes), instead of just 252 bytes.
- If the lencode is 254, then this extended length is 4 bytes long, and therefore allows the TLV value to have up to 2^{32} bytes (4 gigabytes) of data.
- Lencode equal to 255 means the length extension is 8 bytes long, making it possible for the value to contain 2^{64} bytes of information (16 zetabytes, which is roughly 1.76×10^{13} gigabytes).

The structure of NDN packets is well defined (see [4]). A parent TLV, called $\mathbf{TLV_0}$, is the sole container for the entire packet payload. Its type is Interest (0x05) or Data (0x06). TLV_0 inholds all of the NDN packets' structure according to the type. In particular, the first child TLV (in either case) is always $\mathbf{TLV_N}$, which is the TLV whose children are the name components. Each component is also a TLV (see Figure 2.6).

[4] Minus one.

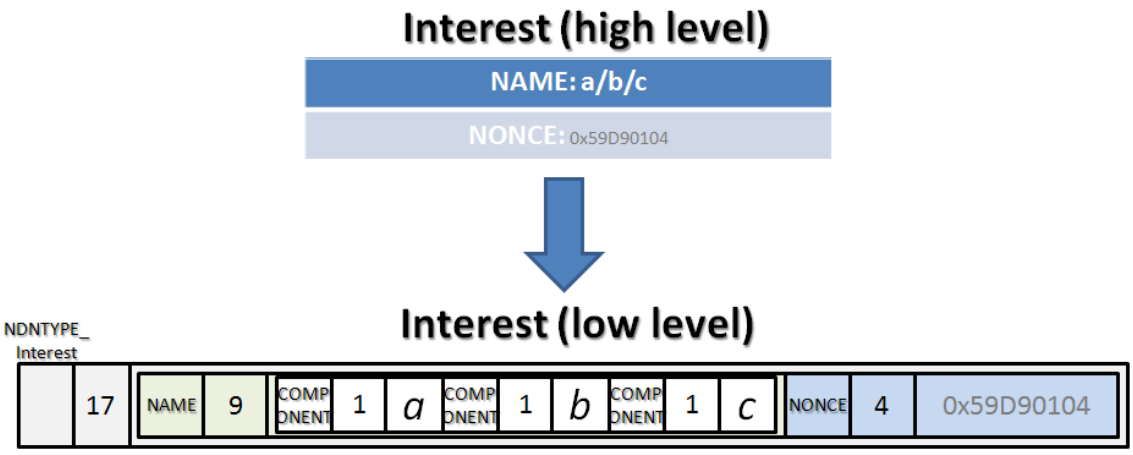

Figure 2.6: High-level perception of an NDN packet and its encoding as TLV.

2.2 Programmable Networks

Software-defined Networks [22] were the first instance of programmable network that has reached to production. SDN enables control plane programmability, but the data plane maintains itself fixed, as common switches and routers are fixed-function, in order to operate at the required very high speeds. Recently, however, the market has seen the emergence of programmable switching chips [18]. Despite being programmable, these devices operate at terabit speeds [18].

SDN separates the control plane (which manages the forwarding tables) from the data plane. This enriches network flexibility, enabling easier configuration, bandwidth allocation and security enforcement. The data plane components become mere packet-forwarding devices, relying on the controller to setup route state by means of (usually) the OpenFlow protocol [22][23].

OpenFlow started by supporting a small set of protocols. As new use cases started arising, this set has expanded version after version. Nevertheless, it remains rigid. An operator who desires to run a very specific protocol in his company's network cannot make use of OpenFlow if it does not support that protocol. Faced with this limitation, and with the availability of programmable switches, P4 [17] was proposed as a high-level language to program network devices, endowed with three great advantages.

1. **Reconfigurability:** The controller[5] can redefine packet parsing and processing on the field.

2. **Protocol Independence:** The language is flexible and unconstrained to specific protocol headers. On the contrary, P4 permits one to define his own headers and how the packet should be parsed and processed.

[5] Although the paper mentions a controller, using P4 does not make it mandatory to have one on the network.

3. **Target Independence:** Programming a switching chip using a low-level interface is similar to microcode programming, and nontrivial. A P4 program, however, can be written with no regard for the end device. This is in all similar to a C program written obliviously with regard to the underlying CPU architecture.

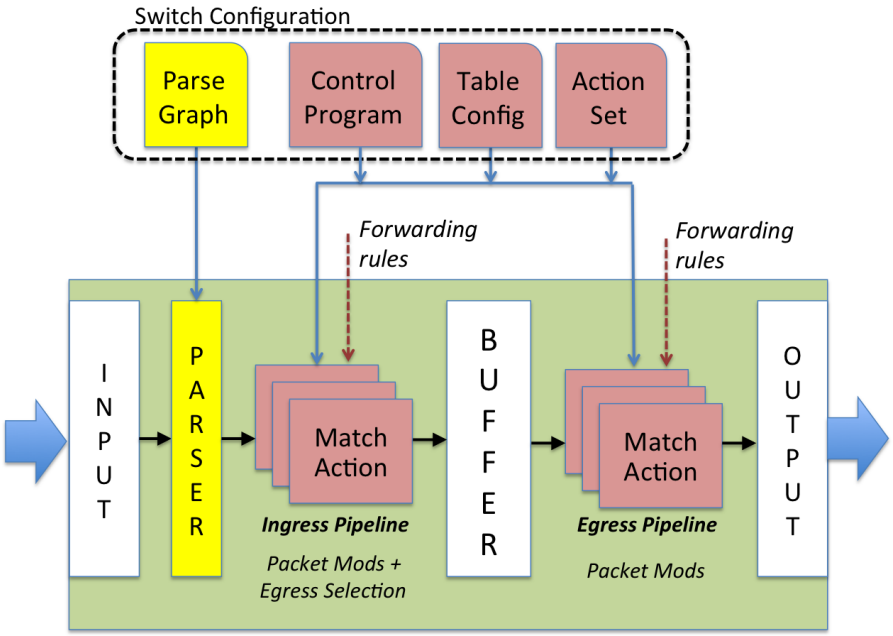

Figure 2.7: The P4 abstract forwarding model [17].

When the packet arrives to a P4-compliant switch, it is handled by the **parser** (see Figure 2.7). Parsing assigns logical labels to physical bit granularity segments of the packet. When the packet arrives to the **ingress** and **egress** processing pipelines, it uses those logical names, chosen by the programmer, to refer to the packet fields. For example, imagine the parser begins by extracting an Ethernet header, which we decided to call "eth". Suppose we defined that header as having three fields – the first two with 48 bits and the third with 16 bits, named "etherType". From thereon, if we write `eth.etherType`, the P4 switch knows it's a reference to bytes 12 and 13 of the physical packet.

In turn, the processing pipelines, ingress and egress, run the packet through a series of match+action tables. These tables can, among other things, modify header fields, set their output ports, or even clone the packet [3].

As a high-level language, P4 must be compiled down to the switch target. The P4 Consortium provides a front-end compiler that analyses the program for syntax correctness and transforms it to an intermediate representation (IR). The device manufacturer, in turn, must provide their back-end compiler to transform the IR to the device's own internal language (see Figure 2.8).

The original paper mentions a **table dependency graph (TDG)** that we deduce comes from the IR. As the name suggests, dependencies between tables constraining parallel

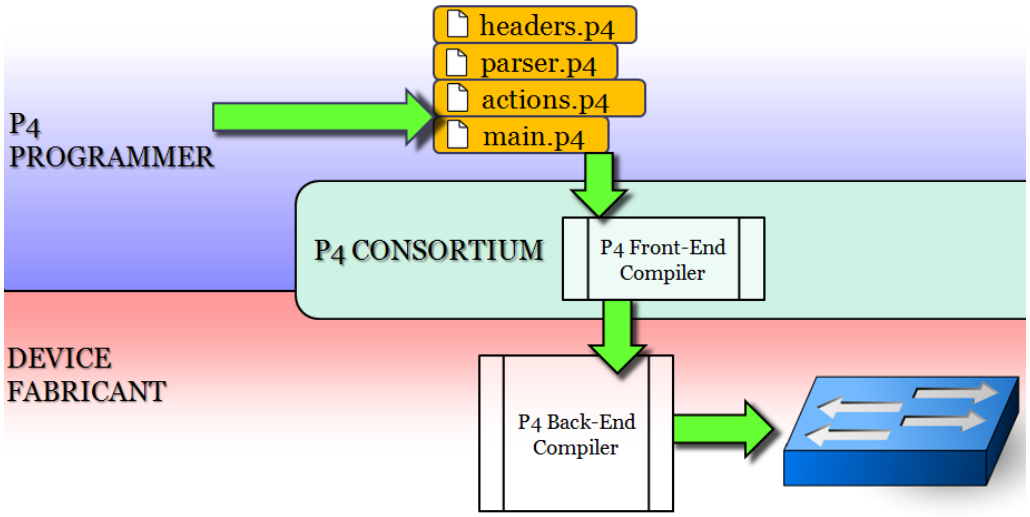

Figure 2.8: Workflow of a P4 programmer.

execution can be inferred using the TDG [17].

P4 has already gained notoriety [2] and is supported by companies such as Google, Amazon, Intel and Microsoft, among others. A number of works have already used the language, such as [27].

2.2.1 P4-14 vs P4-16

The previous version of the P4 language, P4-14, assumes a specific architecture composed of the parser, ingress and egress processing pipelines [3][17], as in Figure 2.7. P4-16 does not make this assumption, supporting any architecture the manufacturer wishes to describe in an **architecture description file** [8].

Logically, since it supports any architecture, we can write one such file to describe the architecture P4-14 assumes by default. This has already been done by the P4 Consortium, in a file called `v1model.p4`. We henceforth refer to this architecture as v1model. It is the only architecture supported by the P4 software switch [1]. It provides a set of primitives and useful stateful memories, like the counters and registers.

The advantage of supporting multiple architectures is that manufacturers can now expose more capabilities of their device to differentiate. In addition, P4-16 restricted the set of primitives enabling it to promote backwards-compatibility. As such, it is expected this current release of the language to quickly become the reference.

This is the main reason we chose P4-16 to implement our NDN router. But there were other motivations as well. First, the previous work on P4 for NDN, described in §2.3, uses P4-14 [26], somewhat constraining this solution. For example, the maximum number of NDN name components is hardwired, due to P4-14's intrinsic nature. An expansion of the max components is therefore a challenging task, involving hand-writing code to support that increase of maximum components. P4-16, on the other hand, comes with numerous

parametrization features by default.

Second, and following from the above, we wished to make our solution as generic and modular as possible. As P4-14 constrains register sizes to a specific bit width [3], this would compromise storing Data content (in the Content Store), that does not have a fixed size, for instance.

Third, and most importantly, as P4 enables new switch architectures, it gives flexibility to innovate without the constraints of P4, by means of extern functions. We believe this to be crucial to materialize, in practice, fundamental structures such as the content store, an option we consider in our work.

2.2.2 P4-16 program example for the v1model architecture

During our brief overview of P4 (§2.2), we learned that we could program devices using this high-level language, provided they are compatible, by offering a back-end compiler to translate an intermediate representation to the device's own language. This section surveys the features P4 provides. We construct a very simple switch that parses and forwards only the IP datagrams it receives.

Data: headers and structs

P4 programs begin with **header** definitions. They are defined similarly to C structs, indicating any number of fields sequentially aligned in memory. P4-16 also allows the definition of **struct** for various purposes, such as user metadata, the logical, parsed packet representation, temporary variables, among others. When working with this architecture, the user is obliged to use them to at least define the parsed packet (see Listing 2.1) and the user-defined metadata (not represented).

Listing 2.1: Header and struct definitions in P4-16.

```
header ethernet_h {
  bit<48> dstAddr;
  bit<48> srcAddr;
  bit<16> ethType;
}

struct Parsed_packet {

  ethernet_h ethernet;

}
```

Parser

Just as we parse a string in Java, for example, to grab an integer, we parse the packet in P4 to attempt to assign logical names to physical portions of the packet. Namely, the headers.

Listing 2.2: Parser definition in P4-16.

```
parser TopParser(packet_in b, out Parsed_packet p, inout
    Metadata m) {
```

```
    state start {
      b.extract(p.ethernet);
      transition select(p.ethernet.ethType) {
        0x800: accept;
        _: reject;
      }
    }
  }
```

Listing 2.2 shows an example of a parser. It starts by extracting an Ethernet header, as defined in Listing 2.1, from the physical packet, represented by the object `packet_in`. It is thus assumed that this is the first header of the packet. Then, it checks bytes 12 and 13 using the logical name `p.ethernet.ethType`. If it finds the value 0x800, then the packet is assumed to be an IP datagram, and proceeds to the processing pipeline. Otherwise, it is rejected and dropped.

Processing pipelines

The v1model architecture declares ingress and egress pipelines for packet processing, so we're going to define them. After parsing, our IP datagram follows to the ingress pipeline. Now, we need to read the destination MAC address and set the output port accordingly.

Listing 2.3: Ingress and egress definitions in P4-16.

```
control TopIngress(inout Parsed_packet p, inout Metadata m,
    inout standard_metadata_t stdm) {

    action Drop()
    { mark_to_drop(); }

    action Set_Egress_Spec(bit<9> port)
    { stdm.egress_spec = port; }

    table egress_arbitier {
      reads = {
        p.ethernet.dstAddr : exact;
      }
      actions = {
        Set_Egress_Spec;
        Drop;
      }

      default_action = Drop;
    }

    //The actual flow of execution begins here.
    apply {
      egress_arbitier.apply();
```

```
    }

}//End of ingress pipeline.

control TopEgress(inout Parsed_packet p, inout Metadata m,
    inout standard_metadata_t stdm)
{ }
```

Listing 2.3 shows the processing flow. The ingress pipeline, which we name `TopIngress`, acts upon a parsed packet, the user-defined Metadata, and the standard metadata. Since they all have **inout** status, we can read from and write to all of them. Previously, we saw the `Parsed_packet` structure as **out**, which is a status of a parameter that has not been initialized. **in** means read-only.

The ingress processing flow is made of a single call for `egress_arbitier` to be applied. This table reads the parsed packet's ethernet destination address and executes one (and only one) of two actions: drop or set output port. Whether one or the other occurs depends on what entries are there to match on. Remember P4 is meant to define only the data plane. It can read from tables, but it cannot modify them. Tables are populated at boot and/or run time by the control plane.

If we find a matching entry whose action is `Set_Egress_Spec`, then we modify the standard metadata egress_spec field to indicate that the packet should follow through that port.

After we've switched the packet appropriately, there's nothing left to be done. Therefore, our `TopEgress` is empty.

Other blocks, data types and features

The architecture offers three other control blocks: the deparser, the checksum calculator and the checksum verificator. The deparser is a novel feature in P4-16 which allows the programmer to selectively deparse headers, omitting or rearranging them. In principle, we should write `b.emit(p.ethernet)`, indicating that we wish for the Ethernet header to be returned to the packet and transferred back to the network with it.

Throughout our demonstration, we encountered the **bit** type multiple times. This datatype represents an unsigned bit vector of whatever length known at compile-time. There are other types available, such as **int**<x>, which is a signed integer, or **bool**.

There are other constructs in the language which deserve mentioning[6]:

- One of the most important is perhaps the **varbit** datatype, which is a variable length bit string. There are two catches to using this datatype. First, when declaring

[6] Throughout this text, the reader may encounter words like **table**, **parser**, and **header_union** in bold and colored blue or teal. These are P4 keywords, not emphasis.

a **varbit**, it is mandatory to specify its maximum width, a compile-time known constant. For example, **varbit** $<$ 2048 $>$ means the field is at most 2048 bits long.

The second catch to using varbits is that the language offers no ways to interact with them, other than: 1) extracting packet data into them; 2) emitting them in the deparser block; 3) copying over the data to a varbit of the same maximum size, and 4) hashing the value held by the varbit through the primitive action or P4-16's **extern** equivalent. Therefore, information can be stored within varbit, but, once there, it is generally inaccessible for reading or processing.

- Both versions of P4 also possesses the notion of **header stack**, which is no more than an array of headers of the same type. It is useful to deal with layers of MPLS, for example.

- P4-16 has a **header union**, which is a data type paradigm similar to C unions. The header union is defined as a set of headers, and only one of them is filled at all times. This is useful when we encounter a crossroads between several possible header shapes (for example, TCP options). Unlike C unions, however, they do not have underlying memory quirks, behaving more like tagged unions [11].

- Novel to P4-16, and perhaps most important of all, is the **extern**. These are constructs implemented by manufacturers directly on the device, while their interface is specified using P4 syntax in the architecture description file. This permits vendors to expose more functionality. For the software switch, implementing an extern could imply creating a C++ class.

- P4-14 provided stateful memories such as the counter, the meter and the register to maintain inter-packet state. When P4-16 was released and P4-14's architecture became v1model, these features were declared as **extern** in its architecture description file.

  Externs come in two flavors: extern functions and extern instantiable blocks. Functions are invoked in the way you'd expect to do whatever work they're programmed internally to do. Instantiable blocks introduce P4 to the object-oriented paradigm. Their interface provides a series of methods and may contain generic types. They are instantiated inside the P4 program and bound to a specific P4 datatype.

Although the program we presented is syntactically correct, and is accepted by the front-end compiler, the software switch back-end compiler rejects it due to its limitations. We explore those limitations later in §4 (Implementation). The reader interested in further exploring P4-16's capabilities may read the full P4-16 specification in [8].

2.2.3 Behavioral Model 2 (BMv2) and p4c

There is an implementation of a P4 software switch called Behavioral Model 2 that almost completely supports both P4-14 and P4-16 programs, though prepared for P4-14's default architecture, `v1model.p4` [1]. We conclude this analysis with a brief overview of BMv2, also known as the "P4 software switch" [1]. This analysis is important so that we can understand the terms, as well as the differences between what the P4 language defines and what the current software switch supports.

First, it is important to make a distinction. BMv2 itself is a C++ library. It possesses an object model that seems familiar to our context. Its classes include Packet, Header, Field, Data, and so on. See Figure 2.9 for a simplified version retaining the part that concerns this work.

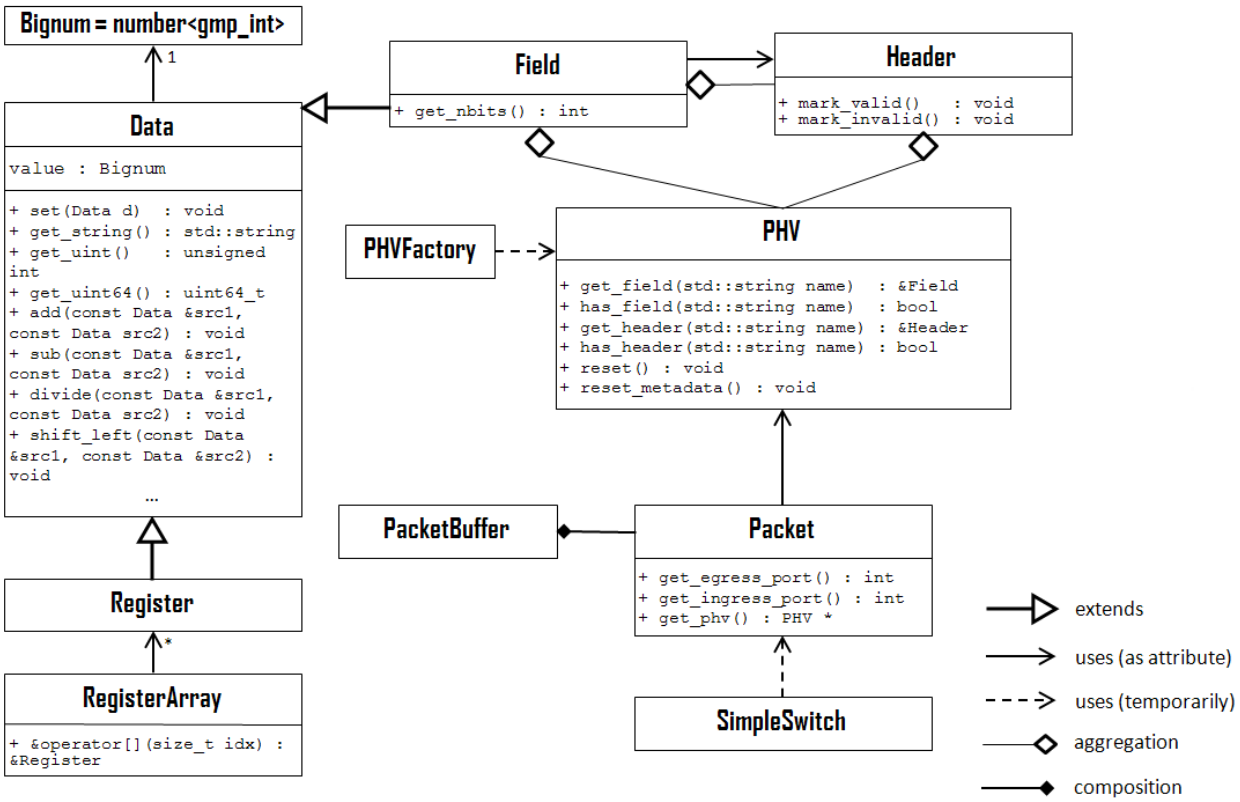

Figure 2.9: A loose, very simplified excerpt of the BMv2 class diagram.

BMv2 also has a directory named **targets**. These are, in fact, the runnable components. Each of these targets implements the P4-14 specification [3], in software, to a different extent. For example, `l2switch` implements only three or four primitives, using the object model provided by the BMv2 library.

Of the three targets provided by the P4 Consortium, the most complete is `simple_switch`. However, it does have limitations, some of which we ran into during our work. We explore them in §4.1.

As mentioned in §2.2, the p4 compiler is divided into front-end and back-end. The front-end verifies the program's syntax and semantics to conform to the specification. This exempts manufacturers from having to repeatedly build a compiler for syntax checking. The back-end converts the IR to something the device can understand. BMv2 (in particular, simple_switch) is a software target, and therefore has a back-end compiler for it as well. P4-14 and P4-16 programs can be compiled for it using p4c [12], the official, yet still alpha-stage, p4 compiler. One of its runnables is `p4c-bm2-ss`, which, as the name suggests, is the back-end compiler for BMv2 simple_switch.

2.3 NDN.p4

NDN.p4 [26] was the first attempt to define an NDN router's data plane as a P4 program, and it is therefore the work closer to ours. This work proposes a basic version of the PIT and the FIB. The main limitations of this solution is that it does not include a content store, vital for the success of the NDN as a distribution network, and, by using P4-14, its flexibility is limited, and real deployments are not to be expected in practice.

We study NDN.p4's implementation in this section.

2.3.1 Type-length-value parsing

NDN.p4 begins by defining five header types[7]. The first has 8 bits of `type`, 8 bits of `lencode` and a variable amount of bits for the `value`. The other headers have 16, 32 or 64 bits of extended length, but no variable field for the value. These are called "TL" headers, meaning they represent only the type and length.

When parsing, NDN.p4 begins by looking 2 bytes ahead. Scanning the last of those two bytes, it decides what header should be extracted. Concretely, it jumps to one of four states: one extracts the TL header which has 16 bits of extended length, the other extracts the header type with 32 bits of length extension, and the last extracts the header type with a 64 bit extension. By default, it extracts the header containing 8 bits of type and 8 bits of length (see Listing 2.4).

Listing 2.4: Parsing TLV0 in [26].

```
parser parse_ndn {
    return select(current(8,8)) {
      253 : parse_medium_tlv0;
      254 : parse_big_tlv0;
      255 : parse_huge_tlv0;
      default : parse_small_tlv0;
    }
}
```

[7] Keep in mind this solution was written using P4-14, which syntactically is quite different from P4-16.

TLV_0 and TLV_N are the only ones that can have this extended field (and no value, since they are parent-TLVs). For all other NDN packet fields except content, NDN.p4 assumes a small TLV; that is, of up to 252 bytes of value.

The Data content is not treated. The parser extracts the type, the length and the extension into a TL header, but does not parse any content value whatsoever. This seems to hint that NDN.p4 is therefore unfinished with regard to basic Data packet processing.

2.3.2 Forwarding Information Base (FIB)

P4 does not support longest prefix match on strings. To workaround this, [26] backs up on two other match types: exact and ternary.

First, the router hashes aggregations of components. More concretely, given a name made of components c_0, c_1, ..., c_n, the router employs a hash function h (in practice, crc-16) to calculate h(c_0), then h(c_0, c_1)... and finally h(c_0, c_1, ..., c_n), where n is the number of components. It does this both when inserting route entries into the FIB and when an Interest is received.

- When the control plane inserts a route, it has to add a number of entries to the FIB given by $Max - n + 1$, where n is the component count of the name being inserted and Max is the maximum number of components the device is programmed to parse. The entries are inserted with the concatenation of all hash outputs and with a number of components that goes from n to Max (see the layout of Table 2.1 for an example).

- When an Interest is received, the router calculates the aggregations as mentioned above. Then, it performs an exact match on the number of components and a ternary match on the aggregation with the right number of components.

Considering an example where the router receives an Interest for "a/b/c/d" and the FIB contains two routes as displayed in Table 2.1, then the Interest still matches two entries from those two different routes (highlighted in yellow on the table). The number of components matches (4 components). Also, the router calculated h("a/b"), yielding BF41, which matches the third rule, and h("a/b/c"), yielding 357F, which matches the fifth rule.

Route	Ternary Match (hex digits)	ncomps	Egress Port
"a/b"	0x ???? BF41 ???? ????	2	13
"a/b"	0x ???? BF41 ???? ????	3	13
"a/b"	0x ???? BF41 ???? ????	4	13
"a/b/c"	0x ???? ???? 357F ????	3	21
"a/b/c"	0x ???? ???? 357F ????	4	21

Table 2.1: A FIB in NDN.p4 containing 2 routes, one for "a/b" and another for "a/b/c". An Interest for "a/b/c/d" matches two entries. $Max = 4$

Recall our objective is longest prefix match, so the bottom rule should be chosen, and the Interest should exit towards port 21. To disambiguate, NDN.p4 adds an extra priority field to the FIB (not shown in Table 2.1), which is set higher for routes with a bigger number of components.

2.3.3 Pending Interest Table (PIT)

The PIT was built as a set of 16 bit hashes. When an Interest arrives, the router consults the PIT to see if its set contains the name hash. If it doesn't, then the Interest is forwarded out the corresponding output port.

Looking into the source code, one can see the authors meant to implement Data multicast by relying on a bit mask[8]. The compiler would then be modified to replicate the code in order for the Data packet to be copied and sent out in accordance to that bit mask. However, the lines that were supposed to be part of that process are commented out. Thus, multicast of Data is not supported.

In its latest setup, NDN.p4 stores no port upon processing an Interest. When receiving a Data packet, the action is to send it out a fixed port that was determined in a file that populates tables at boot time. This is another limitation: Data packet forwarding does not happen correctly.

2.4 Summary

In this chapter, we explored two proposals to change the network that were motivated by the problems of IP, namely, named data networks (NDN), in 2.1, and programmable networks, in 2.2, both of which provide the core context to our dissertation. We finished by exploring NDN.p4 in 2.3, which is the main background work on which ours improves.

[8] We employ an identical scheme, though we arrived at it independently.

Chapter 3

Design

In this chapter we discuss our solution design to meet the objectives proposed at the beginning of this dissertation. It would be uninteresting to be exhaustive about possibilities when, in the end, we would have to adapt the solution to the constraints imposed by P4 and programmable switches in general. Therefore, bear in mind these algorithms were envisaged taking P4's capabilities into account.

This chapter is organized as follows. We describe the general architecture of our solution. Afterwards, we introduce the reader to the partition structure (§3.1), which is key in our improved FIB design. Then, after an overview of the processing flow (§3.2), we study our design of the FIB lpm matching (§3.3), the PIT (§3.4) and the Content Store (§3.5).

We strictly follow the name data networking architecture as presented in Figure 2.4. As we have mentioned, our solution is the first to design and implement a full NDN router with all its core functionalities, so the building blocks faithfully abide to those represented in the figure.

3.1 Partition

Recall from §2.1 that NDN uses arbitrarily long names, and not 32-bit addresses, to route Interests, obeying the longest prefix match (lpm) rule applied at the granularity of the component. For example, when the router receives an Interest for "*pt/ulisboa/fciencias/index.html*", it should prefer a route for "*pt/ulisboa/fciencias*" over a route for "*pt/ulisboa*". Interests seen by the router for the first time are forwarded using the structure known as the FIB, which maps names to output interfaces.

Any solution envisioning the use of (variable-length) strings cannot be materialized directly because P4 offers little support for fields of arbitrary length. P4 registers and tables do not accept, respectively, storing/reading or matching on arbitrary length fields (**varbit**). Therefore, any solution is required to map strings to a construct of a width deducible at compile time. This is an intricate limitation of *any* P4 implementation. NDN.p4

[26], for example, leveraged on P4's capability we mentioned earlier of making it possible to hash variable length fields.

3.1.1 Motivation for a new solution

NDN.p4's [26] method backed up on ternary match of an aggregated hash of components and exact match on the number of components. The authors set the maximum number of components to 4, and the FIB was defined as a table that made: an exact match on the number of components, and 4 ternary matches on the following fields: h(c_1), h(c_1, c_2), h(c_1, c_2, c_3)), where c_i is component number i, and finally the hash of the whole name.

To insert a route onto the FIB, the control plane should add $Max - n + 1$ rules, where $Max = 4$ and n is the number of components of the route name. NDN.p4 [26] uses crc-16, therefore its hash length is 16 bits.

So, each rule necessarily has at least 4 fields (not counting priority or number of components), of 16 bits each. If there are two components, then one needs to add $4 - 2 + 1 = 3$ rules. In total, this amounts to $16 \times 4 \times 3 = 192$ bits.

The resulting size of adding so many rules does not scale well. For a 25 maximum component device, a route of 19 components, and a hash function of 16 bit output, a route occupies at least $25 \times 19 \times 16 = 7600$ bits, again not counting the number of components and priority matching fields.

3.1.2 An innovative concept: the partition

Recycling Signorello et al's [26] idea, we employed hashes as well. Let Max be the maximum number of components the device can parse. Consider a hash function of output length n. We can think of crc-16 ($n = 16$ bits) and 64 maximum components as an example. Given this, we create a structure composed of 64 blocks of 16 bits; generically, in Max blocks of n bits. We decided to call this multipartite structure a **partition**.

> DEFINITION: A **Partition**[1] is a structure divided in a number of equally-sized blocks, where each block i keeps the result of processing component i of a NDN name through a common hash function, or 0 if no such component exists.

The reader can think of the partition as a processed name. Therefore, the PIT, which stored an association of <name, interface list>, stores an association of <partition, interface list> in our solution. The FIB also maps partitions to outgoing ports.

In Algorithm 1, we explain the process to build the partition from the name. It must be used when inserting entries onto the FIB or when matching Interests against them.

Our objective with this algorithm is the following. We hash the first component and place it in the leftmost block; then we hash the second component and place it in the

[1] Throughout this dissertation, we use the terms "partition", "name partition" and "partition of hashed components" interchangeably.

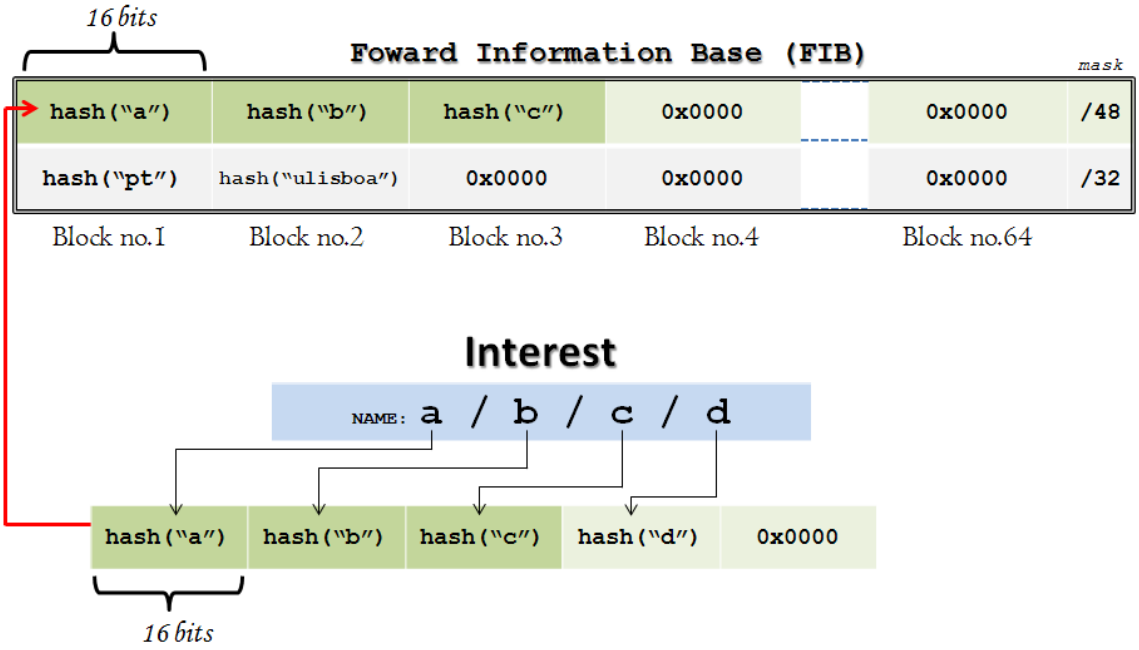

Figure 3.1: The Interest matches a FIB entry by lpm.

second block; and so on. This will later serve to perform longest prefix matching, as per Figure 3.1.

An interesting thing to note about this algorithm is that it has a cycle, but P4 has no iteration or recursion. In any programming paradigm, when the number of instances of some input isn't known beforehand, the solution is to cycle through the input. However, while parsing the TLV structure of the packet, it is impossible to know the number of components ahead of time. TLVs indicate their size, but not the number of children they have. For example, if we find that TLV_N is 6 bytes long, it is impossible to know if it is composed of two components of 1 byte each ("a/b"), or a single component of 4 bytes ("jazz").

One solution to deal with this problem would be to count the number of components by parsing the entire packet once, store in metadata, and then resubmit the packet to the parser with its metadata. This represents a huge overhead; each packet would approximately count for two. Fortunately, the parser is a state machine, and P4 has no problem with a state machine transitioning to itself. Therefore, we wrote a parser state that transitions to itself while we haven't parsed a number of bytes equal to the advertised TLV_N size.

An example

Let us analyze the algorithm more carefully. This algorithm is invoked when we begin parsing TLV_N, whose value is a series of TLVs. As we parse, the first thing we encounter is the TLV_N type and length, as well as the extension if there is one. We remember the

Algorithm 1 Calculation of a partition of hashed components, given a hash function h producing output of length n

```
1:  function BUILDPARTITION(TLV_N)
2:      Read the next component TLV into c
3:      len ← TLV_N.size − 2 − c.extensionsize − c.size        ▷ size is indicated by lencode or extension
4:      acc ← h(c.value)
5:      count ← 1                                  ▷ Names must have at least 1 component
6:      while len > 0 do
7:          if count > MAX then
8:              Abort and signal error
9:          end if
10:         Read the next component TLV into c
11:         acc ← acc ≪ n                                  ▷ ≪ is bitwise shift-left
12:         acc ← acc|h(c.value)                           ▷ Symbol | is bitwise-OR
13:         len ← len − 2 − c.extensionsize − c.size       ▷ c.size includes type, lencode and extension
14:         count ← count + 1
15:     end while
16:     return acc ≪ ((MAX − count) × n)
17: end function
```

size seen ($TLV_N.size$ in the algorithm) so as to know when to stop. We do not know how many components there are, but all names have at least one component, so we parse the next component (line 2) and subtract its total size, including the 2 bytes occupied by its type and lencode, as well as the size occupied by the length extension (line 3, $c.extensionsize$).

We hash the value of the first component TLV and place it inside an accumulator (line 4). The number of components is important to know when the device has reached its limit, but we will need it later as well (line 16).

Let's study an example for an hash algorithm of 16 bit-wide output. Imagine MAX is equal to 4 and the router is going to parse an Interest for "a/b". These components occupy 1 byte each, and their type and length will occupy 2 bytes each. Therefore, $TLV_N.size = 6$. In line 3 we parse the first component immediately, and len becomes $6-2-0-1=3$. The accumulator acc is basically a partition being built. Let us represent its state after line 4 by EMPTY|EMPTY|EMPTY|$h(\text{“}a\text{”})$, where each segment separated by pipe ('|') is a 16 bit block of the partition accumulator (the pipe here is a block delimiter, not bit-OR).

In lines 7 through 9, $count$ is 1, which is less than MAX=4, so we proceed to line 10 and read the TLV that encodes "b". The accumulator is shifted left by 16, becoming EMPTY|EMPTY|$h(\text{“}a\text{”})$|EMPTY. On the next line (line 12), we bit-OR[2] the hash of the next component value, "b", with the accumulator. The accumulator then becomes:

[2] Addition would have worked just as well, but bitwise operations are computationally cheaper.

EMPTY|EMPTY|$h(\text{“}a\text{”})|h(\text{“}b\text{”})$.

Line 13 sets the len variable to $3-2-0-1=0$, so the cycle stops when re-evaluated. Line 14 sets $count = 2$. Our partition building process is almost complete. However, remember that the purpose of the partition was to be a tool for the FIB to achieve longest prefix matching. The significant information covered by the lpm mask is always found on the left side of IP addresses, and the same applies here.

Line 16 performs this last step. Below follow the sequence of performed operations:

$$\begin{aligned} & EMPTY|EMPTY|h(\text{“}a\text{”})|h(\text{“}b\text{”}) \ll ((Max-2)\times 16) \\ &= EMPTY|EMPTY|h(\text{“}a\text{”})|h(\text{“}b\text{”}) \ll (2\times 16) \\ &= h(\text{“}a\text{”})|h(\text{“}b\text{”})|EMPTY|EMPTY \end{aligned}$$

3.1.3 The advantages of using a partition to represent a name

Recall our previous example of a device that parses at most 64 components (therefore, a partition has 64 blocks) and uses crc-16, whose hash output is 16 bits long (each block is 16 bits-wide). Take that example and examine Figure 3.1. It is elucidating on the interactions described in the previous paragraphs. In the figure, an Interest for “a/b/c/d” matches, by lpm, the first entry of the FIB (“a/b/c”). If an entry existed on the FIB for “a/b”, it would be turned down, because the matching entry in the figure has a longer mask.

Note, also, that using a partition, built for a maximum of 25 components and 16 bit hash output, to represent a FIB route would occupy $25 \times 16 = 400$ bits, significantly less (approximately 5%) than in NDN.p4, which, as we said above, occupies 7600 bits.

3.1.4 A note on hash collisions

By now, the reader should be wondering about hash collisions. Informally, it is trivial this problem may occur if we observe that our hash length is only 16 bits (65535 possible values), while the theoretical name length is bounded only by the size of the NDN packet. Logically, if the set of names is larger than the set of hash outputs, then a (total) surjective function exists from the former to the latter; consequently, there are at least two names with the same hash. Let “pt” and “en” be two 16-bit mask entries in our FIB. If, by misfortune, hash(“pt”) == hash(“en”), then the router will be performing wrong forwarding decisions.

The limit case is when the router forwards the Interest back to its incoming port. In a traditional IP network, this would result in sending a packet back and forth until the TTL or hop count reached zero. Fortunately, in NDNs, the Interest nonce would serve for the downstream router to detect that the Interest is repeated, therefore discarding the packet.

NDN.p4 does not address the problem of hash collisions. The authors admit that hash

collisions may occur when registering PIT records, in which case an existing entry is replaced. This is a necessary consequence of the lack of iteration and recursion in P4, as well as the appropriate map data structure. It is not stated what are the consequences of hash collisions on FIB entries, but we assume the aforementioned problem holds.

Because we also employ hashes, our solution can therefore also suffer from the consequences of hash collisions. Since the environment is a forwarding device, we can consider a limited set of options.

- We may use a sophisticated hash (such as a cryptographic one) that is more resistant to collisions, but its use is nontrivial in switches.
- We may increase the hash length to reduce the probability of hash collisions, but we pay for this with increased entry size.
- We may reduce the maximum number of name components, thus reducing table size, but we'll be limiting applications' namespaces.

We took a mental note of these facts to make the resulting P4 implementation (§4) flexible regarding hash length and maximum number of components.

3.1.5 Attempts at collision prevention

For a more formal presentation, let A be the alphabet of possible characters for a name component and $|A|$ its length. For exemplification sake, let A = {a..z, 0..9} $\implies$ $|A| = 36$. Let $c \in \mathbb{N}$ be the maximum number of characters in a component. If our hash is 16 bits long, then

$$\sum_{i=1}^{c} 36^i < 2^{16} - 1$$

is a necessary condition to ensure the possible hash function output is a set large enough to accomodate the possible combinations of characters[3]. Solving this inequation through simple iteration of possible values for c is not too complicated. It yields

$$c \leqslant 3$$

What this result means is that components may have up to 3 characters (amounting to $36^3 + 36^2 + 36 = 47988$ string combinations, which is less than $2^{16} - 1 = 65535$ hash outputs). Examples: "www/cam/ac/uk", or "www/fc/ul/pt" (all components have no more than 3 characters, and all these characters are in A).

Generalizing from the formula above,

[3] This does not prove there are no collisions; that, of course, depends on the hash function, but the contrary condition ensures that some 2 strings share the same hash output.

$$\sum_{i=1}^{c} |A|^i < 2^h - 1$$

(h is the hash length) is a condition that ensures the set of hash outputs is larger than the possible string combinations for a component.

3.2 Packet Processing

Algorithm 2 describes the processing flow we want NDN packets to undergo. Read below for a description of the algorithm.

Algorithm 2 Flow of processing for a packet

```
1:  function PROCESS(packet)
2:      p ← BUILDPARTITION(packet)                          ▷ See Algorithm 1
3:      v ← PIT.get(p)          ▷ The bit vector for this name is retrieved into v
4:      if packet.type = INTEREST then
5:          d ← CS.retrieve(p)                              ▷ CS is the Content Store
6:          if d ≠ NULL then                  ▷ CS had cached Data for this name
7:              Serve d through packet's ingress port
8:              Drop
9:              return
10:         end if
11:         UPDATEPIT(p)                                    ▷ See Algorithm 3
12:         if v = 0 then                      ▷ First Interest seen for this name!
13:             Run packet through the FIB                  ▷ Sets output port
14:         end if
15:
16:     else if packet.type = DATA then
17:         if v = 0 then          ▷ Empty port vector means this Data is unwarranted
18:             Drop
19:             return
20:         end if
21:         multicast(packet, v)
22:         PIT.clean(p)
23:     end if
24: end function
```

Lines 4 and 16 check what type of packet we're dealing with.

- If we're dealing with an Interest (line 4), then we check the Content Store for Data (line 5). If we find Data cached for this name/partition, then we can serve directly (line 7) and drop the current packet (line 8). Otherwise, we update the PIT to record the partition (line 11). This call will set the bit corresponding to the port whence

this Interest came. If, before that update, the bit array (extracted in line 3) was 0, then we must forward it upstream (line 13).

- If we're dealing with Data (line 16), then we check the partition we loaded to see if it is non-empty. If it is empty (line 17), then this is a spurious Data packet and should be dropped (line 18). Otherwise, we multicast it to all set ports (line 21). Afterwards, we can clean this entry (line 22).

Intentionally, we do not detail what happens in the ingress and what happens in egress. The call to *multicast* can be thought of as a two-phased wrapper for Algorithm 4 and Algorithm 5, which, as we'll see, are quite similar in design.

3.3 FIB Longest-prefix Matching

Our NDN forwarding information base is similar to that of a regular switch. Instead of holding 32-bit addresses, it holds partitions. Interests that need to be forwarded have their name calculated into a partition, which is then matched against the FIB. If an entry matches, then the Interest goes out the associated output port.

3.4 Pending Interest Table

In section §2.1 we learned the Pending Interest Table, abbreviated PIT, is a structure that stores a mapping of names to list of interfaces. When an Interest is seen and no Data exists in the Content Store to satisfy immediately, the input interface is recorded on the list for that name. When Data arrives from upstream, the packet is sent to all listed interfaces.

In this section we explore our design to achieve these goals.

3.4.1 Record keeping

NDN.p4 [26] stores the name hash and the corresponding input port. We store an association of partition to port bit array/vector, to save memory. The simplest way to do that is to reduce the problem down to the following semantics: either the port receives the Data, or it doesn't.

The rightmost bit corresponds to port 0, the second rightmost bit to port 1, and so on. See Table 3.1 for an example. The bits that correspond to ports 0, 2 and 5 are set ($= 1$), which means Interests for "a/b/c" have been received from them. When Data for "a/b/c" arrives, these ports receive a copy.

PARTITION	PORT BIT ARRAY							
BUILDPARTITION("a/b/c")	0	0	1	0	0	1	0	1

Table 3.1: A port bit array of 8 positions (meaning the device has 8 interfaces).

When Data arrives to satisfy an Interest, we build the partition from its name. If the PIT contains a record matching that exact partition, we load the corresponding bit vector; each port whose bit is set receives a copy of the Data.

When the first Interest for a given name is seen by the router, its hashed components partition is stored alongside the port bit vector with the corresponding ingress port bit set. As Interests for the same name are received, their ingress ports are added to the bit vector as per Algorithm 3. Updating the PIT is trivial. We retrieve the current port array (line 2) and we bit-OR it with 1 shifted left by the ingress port (line 3). For example, bit number 5 (counting from the right) has the value $2^4 = 16$. If the ingress port is 5, the result on line 3 of the algorithm is $1 \ll 5 = 16$.

Algorithm 3 Procedure that updates the PIT when an Interest arrives

```
1: function UPDATEPIT(partition)
2:     portvector ← PIT.get(partition)        ▷ Returns the current vector for this name
3:     portvector ← portvector|(1 ≪ stdmetadata.ingressPort)
4:     PIT.store(portvector)
5: end function
```

3.4.2 Data multicast

Data multicast is necessary in NDN to satisfy multiple Interests for the same resource at once. Instead of just switching the Interests, the router sends only one upstream, and, when Data flows back, it is multicast to all interfaces that requested it.

Multicasting Data has an inherent implication of creating multiple packets to send to multiple interfaces. P4's only way to achieve this is by using the clone primitives. The challenging part is these primitives work in a deferred fashion. Meaning, the packet is cloned only at the end of ingress or egress, and not immediately upon invoking the function.

In v1model, a packet is first processed by the ingress pipeline, then following to egress. Algorithm 4 prepares the packet for multicasting at ingress. It receives a Data packet (line 1) and fetches the corresponding bit array (line 2). A bit array equal to zero means all bits are zero; therefore, no one requested this Data. Lines 3–7 drop it if that is the case. Otherwise, we know that some interface requested this Data, which means some bit is set.

stdmetadata is a predefined **struct** in the v1model architecture. It is shared between the device and the P4 program and is tied to a packet being processed. The P4 program can control, among other things, what happens to the packet, such as setting its outgoing port, by modifying `egress_spec`. We stress the difference between this standard metadata and "regular" metadata, which is defined by the P4 user at his liking (for example, to hold our partition or port bit vector).

Algorithm 4 Procedure run in the ingress pipeline to prepare a Data packet for multicast

```
1:  function INGRESSPREPAREMULTICAST(data)
2:      portvector ← PIT.get(buildPartition(data.name))
3:      if portvector = 0 then                          ▷ No one requested this Data
4:          Drop                                        ▷ This is spurious Data, let's drop it
5:          return
6:      end if
7:      stdmetadata.egressSpec ← 0                      ▷ Decides the egress port for the packet
8:      while (portvector & 1) = 0 do                   ▷ & symbol is bitwise-AND
9:          portvector ← portvector ≫ 1                 ▷ ≫ is bitwise shift right
10:         stdmetadata.egressSpec ← stdmetadata.egressSpec + 1
11:     end while
12:     Save portvector to metadata
13: end function
```

$egressSpec$, which determines the output port, is initiated to 0 (line 7). To send the packet in the right direction, it is necessary to find the first port that requested it. This is what the cycle in lines 8–11 does. It shifts the bit array right as it attempts to find a port that requested it. The cycle stops when this bit has been found and the packet's egress port has been correctly assigned.

Let's take a device of 6 interfaces (6 bit array), and bit array 0b010100 as an example. This means interfaces 2 and 4 requested this Data. By the end of Algorithm 4, this port bit array will be 0b000101, and $egressSpec$ will be 2, which is correct.

Algorithm 5 Procedure that clones a packet during the egress processing

```
1:  function EGRESSMULTICAST(data)
2:      Fetch portvector from metadata
3:      portvector ← portvector ≫ 1
4:      if portvector = 0 then                          ▷ A single bit was set
5:          return
6:      end if
7:      stdmetadata.egressSpec ← stdmetadata.currentEgressPort + 1
8:      while (portvector & 1) = 0 do
9:          portvector ← portvector ≫ 1
10:         stdmetadata.egressSpec ← stdmetadata.egressSpec + 1
11:     end while
12:     Store portvector back in metadata
13:     CloneEgressToEgress (include metadata)
14: end function
```

But interface 4 also requested this Data, so we must clone the packet. Algorithm 5 is executed during the egress pipeline and achieves this purpose. Following our example, the Data packet enters Algorithm 5 with bit array 0b000101. Its egress port is 2, as determined earlier.

The port array is retrieved from metadata (line 2) and shifted right once (line 3), becoming 0b000010. When running line 4, we find One bit is still set, meaning some other interface is also requesting Data, so we must clone the packet. Lines 7 through 11 do the same as Algorithm 4: they shift the bit array right until a bit equal to 1 is found. By line 12, $egressSpec$ will have been set correctly.

In our example, line 7 would find 0b000010 & 1 = 0, so the cycle code runs once. It shifts the port vector right to 0b000001 and $egressSpec$ is incremented to 4 (not 3, notice line 7 already incremented it once). When checking the cycle guard once more, we find 0b000001 & $1 \neq 0$, so we exit the cycle. Metadata is updated and the packet is now cloned (lines 12 and 13). Processing terminates for this packet, but the clone restarts the algorithm anew. The packet is submitted to egress port 4 correctly. Its bit vector is 0b000001. When we shift right in line 3, it becomes 0b000000 $= 0$. Therefore, the cycle ends, and this packet goes out of port 4.

At the end of this endeavor, two packets were emitted: one for port 2, and another for port 4, which is in conformity with our initial goal.

If more bits had been set, then the logic would repeat; the bit vector would be shifted right until it found a new set bit, a clone would be triggered, and so on.

3.5 Content Store

In §2.1 we introduced the Content Store as a structure of the NDN router, that allowed it to cache Data packet content, when the producer allows it, by filling the appropriate metainfo. In this fashion, the network itself caches content and brings it closer to its consumers.

The Content Store can be thought of as a mass storage device. It must have an interface for storing and retrieving content. The rest will have to do with the specifics of the target device in question. The challenge posed by content stores is therefore not one of design, but that network hardware, given the current networking paradigm, have limited memory and storage and little support for it.

As expected, P4 is also tailored for the common IP use case, so storing and retrieving just about anything is nontrivial. P4 registers do not store arbitrary length fields, so we devised a simplified version using fixed-length fields. We have designed and implemented two versions of the data store. One uses P4 registers — stateful memory made available for P4 programs. As this memory is limited to fixed-length fields and indexing collisions cannot be avoided, we have devised a second solution that involved implementing the Content Store directly in the switch target. In doing this, we are considering a switch architecture that provides extern primitives permitted by the P4-16 language to make use of the Content Store from inside the P4 program. Details of implementation follow in the next chapter.

3.6 Summary

In this section we explained the design of our solution. After introducing the partition in §3.1 as a central data structure pivotal for our FIB lpm matching process, we overviewed the packet processing flow in §3.2. We then explained how records are kept in the PIT and how multicast is performed in §3.4. Finally, we explored the Content Store in the final section (§3.5).

Chapter 4

Implementation

In this chapter we will discuss the various components of our implementation. Below an enumeration of the limitations we came across, including consequences for our work (§4.1). In line with P4's packet processing flow, we first visit the **parser** to explore TLV parsing (§4.2). We then peruse the **table** definition for the FIB (§4.3), the two methods of constructing the PIT (§4.4), and the two implementations of the Content Store (§4.5).

4.1 Compiler and Target Limitations

To start explaining the present challenges of implementing network solutions in P4-16, in this section we enumerate the bugs we came across and its consequences to our solution. We should emphasis that all functionality we try to use is inline with the P4-16 specification. The problem is its support is still somewhat inadequate.

1. The p4 front-end compiler does not accept the **header union** construct, even though it is now part of the specification. This makes it impossible to parse TLVs with a lencode larger than 252.

2. p4c generates an incorrect .json in certain situations involving the **bool** type [6]. No serious consequences arise; we can use **bit**<1> instead.

3. p4c does not compile subparsers correctly [15].

4. The BMv2 backend compiler rejects **varbit** fields. This interferes with TLV values and forces us to use a fixed-width type for the TLV value.

5. The simple_switch does not accept shifts larger than 255 bits. This hinders the last step of Algorithm 1. It also impedes hash function output lengths from being 256 bits or larger. Shifts may still be stacked to achieve a larger value, but it will have to be hard-coded.

6. The simple_switch cannot access an **header stack** index using a variable, unless that variable is a constant known at compile-time. It also cannot use the hs.next and hs.last expressions to access, respectively, the next free member and the last filled member of the stack. Therefore, access to header stack members have to be hard-coded both in the parser code and in the processing pipelines code.

7. The simple_switch forbids the **extern** function *hash()* from being used in the parser. Recall that using it in the parser was essential due to being the only P4 block where iteration is possible, which is required for Algorithm 1.

8. BMv2 backend rejects explicit parser transitions to the reject state. Simply omitting the reject state turns out to be a mistake, as the packet proceeds to ingress when no state transition is specified. The resolution was to add a metadata **bit** starting at 0. The first action applied in ingress is to drop packets whose metadata bit is 0. At specific parser states, we set this bit to 1 to indicate that the packet should be processed.

All the abovementioned problems implies a loss of our ability to parse an unbounded number of components. Therefore, on our simplified version for evaluation (explained in the next chapter), we had no choice but to severely hard-code most of the parsing and processing flow for a fixed maximum number of components.

4.2 Type-length-value Parsing

The first phase any packet goes through, in every architecture, is the parser. We did not approach TLV parsing on the Design phase as it would necessarily be intimately implementation-specific.

The challenge here was twofold. First, it was necessary to decide what would be the most appropriate header definitions. In ordinary networks, protocol headers come in a sequence: first Ethernet, then IPv4, then IPv4 options if present, then TCP, and so on. MPLS headers may be stacked on top of each other, but they are still linearly disposed. In constrast, parsing the NDN nested TLV structure is not as trivial. This is because the value of a TLV is either arbitrary content or a number of other TLVs, which may themselves be sequentially disposed or further nested.

The second problem is that Length fields, as explained above, may lead to extra bytes standing in-between the Length and the actual Value. Expressing this using P4 and in a practical manner is, again, nontrivial.

Conceiving a solution to deal with the nested TLV structure was pivotal. A linear interpretation is easier to reason about. We discovered this interpretation to be applicable because the NDN packet format is well defined [4]. We can imagine a sequence of octets where the type and length (TL) of TLV_0 come first, then the TL of TLV_N, then a series of

TLV from name Components, and so on (we will see how to deal with extended length later). E.g.:

TLV_0.type	TLV_0.lencode	TLV_N.type	TLV_N.len	COMPONENT.type	COMPONENT.len	(string)

Let us take a look at our header definitions in Listing 4.1, which resemble closely what the reader would expect[1]. In P4-16, widths are generally specified in bits. In that regard, the listing is self-explanatory. The `smallTLV_h` has a Value that may be 252 bytes $\times$ 8 in size, which equals 2016 bits. `mediumTLV_h` is the TLV with up to 2 bytes of extension. Its size may be 2^{16} bytes, which once more must be multiplied by 8, yielding $2^{16} \times 8 = 2^{16} \times 2^3 = 2^{19}$, which, in hexadecimal, is 0x80000. `LargeTLV_h` may have a value of 0x7fFFffFF, which is the maximum permitted by the P4 compiler.

Listing 4.1: Our solution's header definitions.

```
header smallTLV_h {
    bit<8> type;
    bit<8> lencode;
    varbit<2016>
        value;
    //2016 = 252 * 8
}

header mediumTLV_h {
    bit<8> type;
    bit<8> lencode;
    bit<16> size;
    varbit<0x80000>
        value;
}

header largeTLV_h {
    bit<8> type;
    bit<8> lencode;
    bit<32> size;
    varbit<0x7fFFffFF>
        value;
}

header_union TLV_hu {
    smallTLV_h smallTLV;
    mediumTLV_h mediumTLV;
    largeTLV_h largeTLV;
    //hugeTLV_h hugeTLV;
}
```

Normally, a P4-16 parser invokes b.extract(p.header). b is an instance of the `packet_in` structure provided natively by the P4-16 core language and constructed by the device before invoking the parser. This is relatively simple. However, we're dealing with a TLV structure, which has variable length fields, and is arranged in a tree. That is the challenge before us. As our solution, the parser block is composed of a primary parser, as well as a subparser, a P4-16-exclusive feature. The main parser relies on it to extract and fill in every TLV, performing verifications along the way.

Listing 4.2: An excerpt of our subparser.

```
parser Subparser
(packet_in b, inout Metadata m, out TLV_hu TLV) {
```

[1] In practice, when looking at our program on GitHub, you may find tiny variations, as we used `#define` directives to specify a few values.

```
    state start {
      bit<16> len = b.lookahead<bit<16>>() & 0xff;

      transition select(len) {
        253 : parse_medium_TLV;
        254 : parse_large_TLV;
        255 : reject;
        default : parse_small_TLV;
      }
    }

    state parse_medium_TLV {
      //We lookahead for type, length, and extension
      bit<32> extension = b.lookahead<bit<32>>();

      //Keep only the bits that matter
      extension = extension & 0xffff;

      //Remember the extension comes in bytes, so we must
      //multiply by 8 (shift left by 3) to turn to bits
      b.extract(TLV.mediumTLV, extension << 3);

      transition accept;
    }

    ...
}
```

Listing 4.2 is an excerpt of our subparser, contemplating the process that parses a TLV of lencode = 253. The procedure is similar for TLVs of different sizes.

The subparser is given the `packet_in` object — which represents the packet being parsed —, the user-defined Metadata and a TLV that needs to be filled (hence the **out** status). It first relies on the lookahead() method.

To understand this method, the reader should think of the parser as having a pointer in memory that can go forward only by extracting, and never goes back or rewinds. lookahead() permits looking ahead of the pointer without moving it. When the parser looks ahead 16 bits, it will place in variable `len` the type and length of the following TLV, applying a mask to discard the type. It then checks the lencode to decide what header should be extracted, and initiate the appropriate **header union** member.

While examining the above code listings, the reader has probably noticed two shortcomings. First, 64 bit-wide length extensions are not accepted. This is an inherent limitation of P4. `packet_in` offers two versions of extract(): one for fixed length headers, another for variable length headers (implying they have a **varbit** field). Let's analyze the

method signature.

```
pkt.extract(variableSizeHeader V, bit<32> variableFieldSize)
```

As we can see, the **varbit** field's size can be expressed in a maximum of 32 bits. This alone makes 64 bits extensions impossible.

The second shortcoming follows logically from that method signature, but is less obvious. The **varbit** field's size, `variableFieldSize`, is expressed in bits, and not bytes. Logically, then, the case where lencode is 254 has a small restriction. When multiplying by 8 to convert bytes to bits, the 3 most significant bits must be cleared if we are to extract the correct amount (see below).

0	0	0	1	1	0	1	1

$\Downarrow \times 8$ (same as shifting left by 3)

1	1	0	1	1	0	0	0

This introduces a pratical limit to our implementation, where TLV packets may be, at most, $2^{32-3} = 2^{29}$ bytes $= 512$ megabytes in size. This is still plenty and unlikely to be a hindrance, but a limitation nonetheless.

Faced with the above, we use the verify() P4 parser function to ensure that the length extension, when bit-ANDed with a mask of 0xE0 00 00 00, equals 0. Failing to pass this verification implies the packet is dropped.

We finish this section with a small glance of the main parser as it uses the subparser to fill in the various NDN packet fields. As the reader can see in Listing 4.3, our main parser does not extract directly, instead using the encapsulated logic of the subparser for this purpose.

Listing 4.3: A small glance of our main parser (verifications suppressed).

```
parser TopParser(packet_in b, inout Metadata m, inout
    standard_metadata stdm, out Parsed_packet p) {

    Subparser TLVreader(); //Instantiation of the Subparser

    ...

    state ... {
      TLVreader.apply(b, m, p.metainfo);
      TLVreader.apply(b, m, p.content);
      TLVreader.apply(b, m, p.signatureinfo);
      TLVreader.apply(b, m, p.signaturevalue);
    }
}
```

4.3 Forwarding Information Base Implementation

We introduced the partition in §3.1 as a central structure to represent a FIB route. Implementing the FIB per our design in §3.3 is only a matter of correctly building the partition, as detailed in Algorithm 1. The result of translating this algorithm to P4 code is attached in appendix A.1 [2].

The FIB definition, as displayed in Listing 4.4 is close to that of any other P4 table. The key indicates applying this table will perform a match on the packet's metadata, particularly in our hashed components partition. As a consequence, the packet will have its output port set, or it will be dropped.

Listing 4.4: Definition of the FIB for our solution.

```
table fib {
  key = {
    metadata.hashed_components : lpm; //This is the partition
  }
  actions = {
    Set_output_port;
    Drop;
  }

  default_action = Drop; //Not const, therefore the control
     plane can modify it.
}
```

4.4 Pending Interest Table Implementation

As explained in §2.1, when the PIT receives an Interest for a name, it adds the interface whence that Interest flowed to a list of interfaces associated with that name. When Data flows back for the same name, all listed interfaces receive a copy.

Recall from §2.3.3 that NDN.p4 [26] does not store the input port of Interests. It stores a hash output of the complete name on the PIT and consults it upon arrival of Data. If it finds the hash of the name of this Data matches a record in the PIT, then it sends it out a fixed interface defined at boot time.

We implemented the PIT in two different ways: as a **table**, and as a v1model **extern** register. These endeavors are detailed in the following subsections. The table has the advantage that it can be mapped to fast match-action switching hardware, while registers are stateful memories that we can build in the data plane to maintain inter-packet state, and are therefore suitable for fast packet switching, but also offer greater flexibility.

[2] The reader may notice metadata variables such as `m.last_extracted_TLV_size`. These are set by the subparser (we omitted the assignments earlier in Listing 4.2).

4.4.1 Table

Our first implementation of the PIT used the **table**. An important limitation is that P4 allows reading from tables, but not writing to them; updating them is the sole responsibility of the control plane. We created two tables, one named **`pit`**, the other named **`nonces`**. Both store the hashed components partition. **`pit`** stores a bit vector indicating the ports the Data packet should be cloned to when it arrives; **`nonces`** holds pairs of (name,nonce). See Listing 4.5 for our declarations.

Listing 4.5: Our mapping of the PIT into tables.

```
table nonces() {
  key = {
    metadata.hashed_components : exact; //This is the partition
    packet.nonce.value : exact;
  }
  actions = {
    Drop;//Looping Interest
    @defaultonly Send_to_cpu;//Use only in default_action (no
       match found)
  }

  const default_action = Send_to_cpu;
}

table pit() {
  key = {
    metadata.hashed_components: exact;
  }
  actions = {
    Pull_face_list; //Fetches the current bit vector into
       metadata
    @defaultonly Drop; //No PIT record matches this Data
  }

  const default_action = Drop;
}
```

These PIT tables are consulted in two moments: on the arrival of Interest packets and on the arrival of Data packets.

- When an Interest packet arrives, only the `nonces` table is consulted. If there is a (partition,nonce) match, then this is a looping Interest, and is dropped right away. Otherwise, it's a new Interest for a name already seen. The control plane is expected to update both this table (to record the new nonce) and the `pit` table (to record the new port in a bit array) accordingly. Additionally, it will apply the FIB if this is the first Interest seen.

- When a Data packet arrives, we apply() only the `pit` table. If there is no match, then this Data was unexpected, and it is dropped (notice the default action). If there is a match, then we withdraw the current array of set bits. In this case, the control plane should also have deleted the entry both from this table and the `nonces` table.

The main downside of using tables to implement the PIT is that we are forced to store the partition — which has a non-negligible size — twice, effectively implying a spacial overhead by a factor of 2. Moreover, the packet is frequently sent to the switch CPU for the port array to be updated, which comes at a high cost.

4.4.2 Registers

Our second implementation of the PIT is through registers. We defined the PIT as two registers. One is `register<bit<NUMBER_OF_DEVICE_PORTS>>`, where NUMBER_OF_DEVICE_PORTS is a constant. With effect, this is storing a bit vector with as many positions as there are ports. The other is `register<bit<PARTITION_LENGTH>>`.

When adding a record to the PIT, the name partition P is stored into position P % n, where n is the declared number of register positions and % is the remainder operation. The partition is thus stored on the latter register, while its corresponding bit vector is stored on the former. This last register is to ensure that an indexing collision does not alter the bit vector of a different name.

The main advantage of this solution is that the control plane does not have to be involved in the process.

4.5 Content Store

Our work is the first to implement the Content Store in a programmable switch. Two options for its implementation were using **tables** or **extern** blocks. However, the Content Store does not fit the model of a table, not only because tables do not allow variable length fields anyway, but because they are not suitable for storage. We thus opted for the use of externs.

When employing externs, it is necessary to declare its interface. We first used v1model's extern registers, as we did for the PIT. Since P4 registers do not accept **varbit**, this is only possible by hardwiring the TLVs to a specific width. Our testable version considerably simplifies headers — down to a simple TLV of 8 bits of type, 8 bits of lencode and 64 bits of value — and the parser. Registers alone are not to blame for this; recall from the limitations enumerated in §4.1 that we cannot compile programs for simple_switch with header unions or varbits anyway.

This severely constrains flexibility, so an extern defined and programmed directly in the switch is the ideal option. We materialized this second option by writing a sample

Ethernet architecture that extends the v1model architecture with additional primitives, as per Listing 4.6. The listing shows what one could expect to find when working with an NDN architecture willing to offer a Content Store for P4. To properly implement them, however, we would be forced to modify both the front-end and the backend compilers, as they lack support for user-defined externs. Another option, the one we opted for, was to implement the Content Store inside the SimpleSwitch class directly.

Listing 4.6: A possibility for a content store interface.

```
/**
 * Stores Data with identificator 'key'.
 */
extern void cache<K,P>(in K key, in P packet);

/**
 * Retrieves the Content identified by key
 * and returns it in 'content',
 * if identificator 'key' exists.
 */
extern void retrieve<K,D>(in K key, out D content);
```

4.5.1 CS as registers

Similarly to the PIT, CS was implemented as two registers: one to store **bit**<64>, the other as `register<bit<PARTITION_LENGTH>>`, as before. When Data is cached, the content is stored inside the first register, while the associated name partition is stored at the homologue position of the second register.

Once more, the second register is important to avoid committing mistakes when indexing collisions occur. If a name partition P_1 happens to index to a register position already filled by name partition P_2 and $P_1 \neq P_2$, then serving P_2 would be an error.

4.5.2 CS in the switch target

For this purpose, we created a C++ class for the Content Store, in the SimpleSwitch[3]. This target switch was modified to instantiate it with a parameter that is the maximum number of packets the Content Store can hold. Internally, the ContentStore is a map of name partitions to PacketCell. PacketCell is a private class that holds cached Content, envisioning future work wishing to add smart replacement policies to the Content Store. PacketCell can easily be modified to include other parameters, such as the Data freshness period.

The SimpleSwitch was also modified to call the appropriate methods of the ContentStore class at the appropriate times. However, no **extern** calls were realized due to

[3] The code is extensive and not obvious, so instead of listing it here we invite the reader to calmly inspect it on our GitHub repository instead.

lack of compiler support. The result is that the P4 program cannot interact with the Content Store directly, namely to check if it contains Data under some name.

Processing Data remains simple; the Content Store is called only to cache it. This already happens at the end of processing, so moving this to the switch is trivial.

The hard case is processing Interests. The first step is to consult the Content Store for a matching cached Data. But if the P4 program cannot interact with the Content Store, how is this achieved?

Our solution to this problem was to change the P4 program to assume the Data never to be cached. If, by the end of ingress, we find that it is, then:

- The packet type is changed from INTEREST to DATA;
- The name is left unchanged;
- Its Nonce is removed;
- The cached Content is attached;
- Packet length is recalculated;
- And egress_spec (which sets the output port) is set to the ingress port, effectively sending the packet backwards.

Though our goal was proof of concept before efficiency, we note that the only useless work the switch has performed is to update the PIT and run the packet through the FIB (but it is not actually forwarded upstream due to the above challenge).

Not everything can be overridden, and that is what makes Interests a tougher case to handle. Registers' contents (and thus the PIT's) cannot be accessed in run-time. This creates an awkward case. Because the P4 program assumes no Data was in store, it will keep updating PIT records. Now, imagine the Content Store is full and discards one of the cached packets. If it received requests while this content was cached, then its bit vector has been filled. Then, the router interprets this as already having sent an Interest upstream, and does nothing. The consumers are now stuck waiting for a router that will never satisfy their request.

To solve this problem, we decided to change the P4 program running with the modified SimpleSwitch to always run the Interest through the FIB and forward it upstream (when no Data is cached, otherwise the enumerated steps happen).

4.6 History of Development

P4 is a newborn language — the first to enable switch programming. This brings with it huge challenges, but also exciting opportunities. With respect to the former, we faced

many difficulties, in first hand, throughout this year of work: the language, and consequently its support, being re-defined and changed along the way; the existing compilers being developed as we progressed, having as consequence an array of functionality not supported, a series of bugs unreported; etc. On the other hand, it was a privilege to be part of an emergent community that is expected to radically change the way networks operate: sorting out compiler bugs, helping others, etc. Given this not-so-common prospect, we've decided to include an unconventional section with the "history" of our development.

While reading the P4-16 (then draft) specification [5], we devised a first version of our proposal for the "Very Simple Switch" (VSS) architecture. This is a very limited architecture, with a single processing pipeline, providing no stateful memories (registers, counters or meters), and only a single **extern** checksum block.

We later adapted our program to the so-called v1model architecture (which is identical to P4-14) as we realized the P4 software switch was not prepared to deal with any architecture besides the v1model anyway, because supporting a new architecture would require a substantial amount of work that exceeded our tight schedule. Nevertheless, starting with VSS helped us better understand the philosophy behind how P4-16 works.

After adapting our program to the v1model architecture, we wrote a sample architecture description file extending it and providing **extern** functions that, when acting according to the prescribed documentation, would make our program behave as intended. Before we even got to the part of programming them in C++ for the target software switch, the full specification of the language was released in May [8]. A few small details affected the correctness of the P4 programs written for the former specification that had to fixed. Among these changes, there were two that stood out.

1. The *extract()* method could now only take a **header** as parameter, i.e., header fields could no longer be extracted individually. Prior to this, extracting a header could be performed by writing the following series of calls:

   ```
   b.extract(p.ethernet.dstAddr);
   b.extract(p.ethernet.srcAddr);
   b.extract(p.ethernet.ethType);
   ```

   It would be odd to shuffle the order of these calls or omit some of them, given the definition of the Ethernet header. Therefore, we can easily see the logic behind invalidating individual field extraction.

2. The **header_union** construct was introduced. As evoked in §2.2.2, these are C-like unions whose every member is of type **header**, and only a single member is active at all times.

Both these changes invalidated the P4 **parser** implementation we had written, forcing a partial rewrite.

Our schedule was becoming narrow as we frequently found compiler bugs. Time was short, so we decided to jump to evaluation phase in order to verify the correctness of what could already be tested. These phased tests are detailed in §5.

As we progressed through testing, we found several limitations on the `simple_switch` target, which we elaborate in that chapter. Therefore, we were forced to bifurcate our project in two versions. One of them is complete, but cannot be tested, although it would be runnable if both the compiler and BMv2 faithfully followed the P4-16 specification. The other was simplified to abide to all of the limitations we encountered. It can be tested, though at the cost of hindering the flexibility we aimed to achieve.

4.7 Summary

Throughout this section we detailed our implementation. In particular, after an overview of the compiler and target limitations (§4.1, we explained how TLV parsing occurs (§4.2), the FIB definition (§4.3), the two possible materializations of the PIT as a table or as registers §4.4) and our two implementations of the Content Store (§4.5). We closed the chapter by narrating part of our development history (§4.6).

Chapter 5

Evaluation

Because the P4 language, and particularly P4-16, is so recent (draft released December 2016 [5], and v1.0.0 in May 2017 [8], two months before our project delivery), the compiler is still alpha quality (see the GitHub repository README [12]). Therefore, we faced severe difficulties in compiling our programs and running them. Even to test basic functionality, we had to simplify our solution significantly, such as hardwiring value lengths to always be 8 bytes. We have anyway tried to provide an evaluation that tests the fundamental aspects of functionality.

The remainder of this section is organized as follows. First, we mention a few software artefacts that we had to develop as testing tools, in §5.1, due to lack of other support. Then, we describe our test environment in §5.2. We analyze the difference of memory requirements between NDN.p4's solution and our own (§5.3). Lastly, we detail the functionality tests we ran to test our implementation, in §5.4.

5.1 Developed Testing Tools

This section presents some of the tools developed within the project to help us in the evaluation.

5.1.1 rawpkt

As explained, the TLV nested structure does not quite fit the structure of a typical protocol stack. Therefore, we were restrained from using Scapy or other packet generation tools and developed an NDN packet generator, in C. Our software sends out an NDN packet built over Ethernet. The program is made of several modules and compiles to a `rawpkt` executable, which comes with a few handy options:

- `-i <interface>`: Defines the interface through which the packet will be sent out. If not present, "eth0" is assumed as default.

- `-d(eth) <l2address>` Defines a destination MAC address. E.g., `./rawpkt -deth 1:0:aa`.
  - No checks are performed on input other than its length, rejecting anything larger than *strlen*("00:00:00:00:00:00").
  - It is possible to type something like `-d 1:a`, which is equivalent to typing `-d 00:00:00:00:01:0a`.
  - Typing non-hexadecimal characters deterministically produces some hexadecimal value.
- `-n <name>` Sends the packet for this name. We used a default that better suited our testing purposes: "portugal/unlisboa/fciencia/index.ht". Notice all components are 8 characters long.
- `-c <count>` Sets the number of packets to send out. Currently, it corresponds to an iteration of the C syscall *sendto*().
- `-f <file>` Sets the packet as Data and appends a Content TLV to it whose Value is the contents of the file.

5.1.2 makeFIBrules2.py

This is a file based on the one used in NDN.p4 [26], available on Salvatore Signorello's GitHub NDN.p4 repository[1].

Recall that the FIB is built by hashing components in either solution. This script makes it easier to add entries to the FIB, thus avoiding having to rely on online calculators (or otherwise) every time we want to add or change entries.

One may write routes to a FIB.txt file that contains the entries and the respective switch interfaces they are meant to be forwarded to, separated by space. Then, this script, `makeFIBrules2.py`, given the FIB file with option `--fib`, appends the entries to a file specified using option `--cmd`, with the necessary hashing already calculated.

We fetched this script from Signorello's repository and modified it to suit our solution. The original Python script is prepared for adding several entries as prescribed by NDN.p4's ternary+exact match algorithm. For example, if FIB.txt contains the following entries:

```
/portugal/unlisboa/fciencia 1
/portugal/unlisboa 2
/portugal 3
```

Then, for a maximum of 3 components, a HASH_LENGTH of 32 bits, and using

[1] https://github.com/signorello/NDN.p4

algorithm crc-32, the resulting output file using our version of this script will have the following:

```
table_add fib Set_outputport 0xD7AF62AE5B069BA2A833F2EB/96 =>
1
table_add fib Set_outputport 0xD7AF62AE5B069BA200000000/64 =>
2
table_add fib Set_outputport 0xD7AF62AE0000000000000000/32 =>
3
```

5.2 Environment

Our tests were run using a native installation of Mininet [28], a network emulator, and a git-cloned BMv2 in Ubuntu 16.02. We strictly followed every instruction on the GitHub repositories of the behavioral-model [1] and p4c [12].

We also fetched the tutorials repository[2] from p4lang, as it contains a preset environment for testing. In particular, we used P4D2_2017, which is the environment for P4-16. It contains a number of sample P4-16 programs.

The reader wishing to replicate our tests should modify the P4D2_2017/utils/p4apprunner.py to point to a local copy of BMv2. We include the modified p4apprunner in our repository. Scooping the file for "~/PEI/behavioral-model" and "~/PEI/p4c" should yield the correct lines. These should be edited to point to the reader's local clones of BMv2 and p4c, respectively. Keep in mind at all times that all TLVs excluding TLV0 and TLVN must be 8 bytes long.

Mininet is a network emulator and an excelent tool for experimentation. The P4 Consortium modified it to run P4-compatible switches instead of Mininet's default. It accepts the .json file produced as output of compiling a P4 program and behaves accordingly.

Mininet provides a terminal where the researcher can insert commands that change the state of the network or order nodes to do something. Xterms may also be opened on nodes, serving as normal hosts. We leveraged these capabilities to make nodes send NDN packets and capture results using tcpdump.

5.3 Evaluation of Memory Requirements

Given a hash function of length h and a Max name component count supported by the device, the formula to calculate the size used by NDN.p4 to insert a route composed of n components is given by:

$$Max \times (Max - n + 1) \times h$$

[2] https://github.com/p4lang/tutorials

Our solution, on the other hand, has a cost of:

$$Max \times h$$

Both solutions have two common factors, but NDN.p4 needs to add several rules, as many as $Max - n + 1$.

Consider a device that supports 32 maximum components ($Max = 32$). If we were to insert a route with 32 components, the cost would be exactly the same. However, if we were to insert a route with a single component, the difference is enormous. NDN.p4 would spend $32 \times 31 \times 16 = 15872$ bits to hold that route, while our solution would require only $32 \times 16 = 512$ bits, which is approximately 3% when compared to the former value.

Figure 5.1 shows the difference in memory usage between the two methods. NDN.p4's has the same memory footprint for routes with the maximum number of components. However, when the number of components for those routes begins to decrease, the memory occupied substantially grows, illustrating the scalability problem that we've advertised throughout this dissertation. By contrast, our solution's memory footprint remains constant for fixed Max and h.

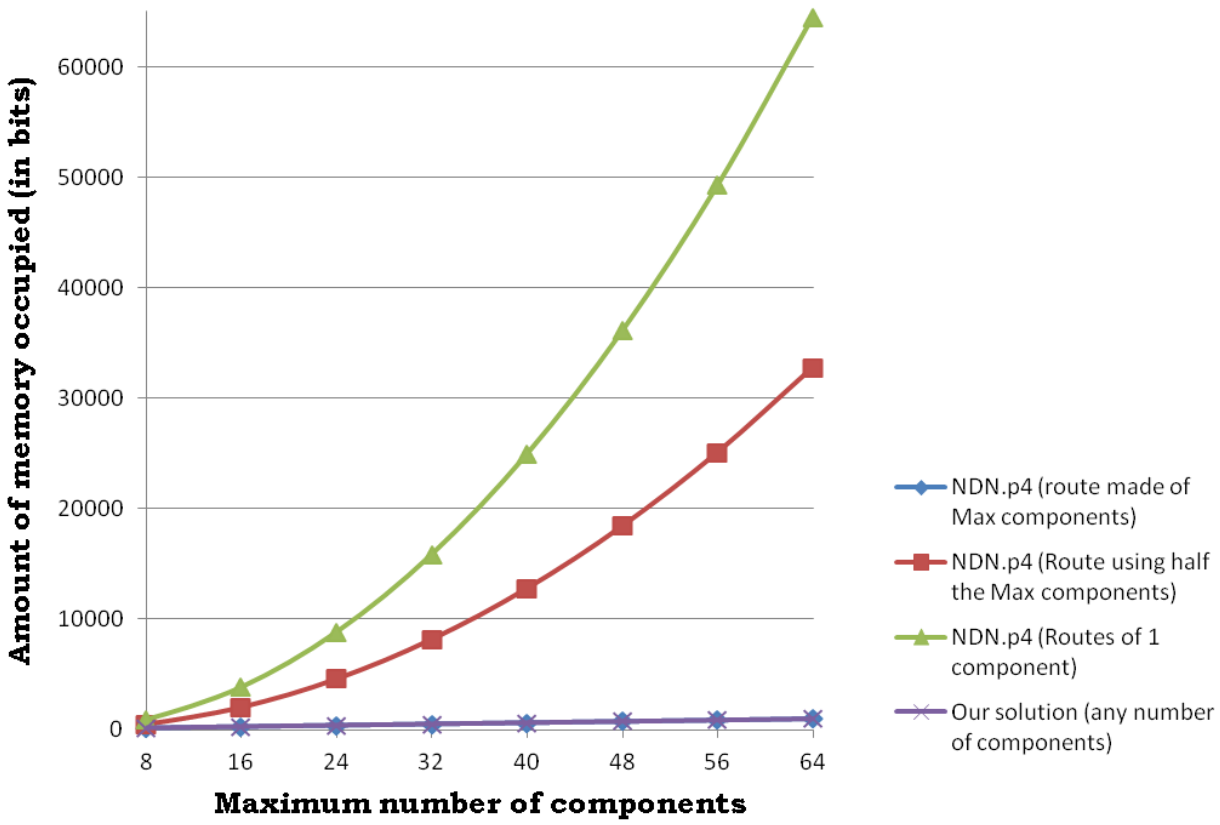

Figure 5.1: Variation of the memory occupied in function of the maximum number of components, with a fixed hash output length of 16 bits.

5.4 Functionality Tests

5.4.1 Parser and deparser

Because `simple_switch` does not accept header unions or varbits yet, we were unable to test our main parser. We thus devised a simplified version of the main parser, called `FixedLengthParser.p4`. This parser only extracts a series of fixed-length TLVs, including up to 4 components.

In the following test, we inserted lines at the end of the final states of the parser to demonstrate that they are reached. If the packet type is Interest, we change TLV_0 type to 0xFE. If it is Data, we change to 0xFF.

1. Parser/Deparser — Test description

SETTINGS

- Simple topology of two hosts connected by a switch running a program with our specified FixedLengthParser and our main deparser.
- Dummy ingress pipeline sends the packet out physical port 1 if the packet came from physical port 2, or port 2 if it came from port 1.
- Empty Egress processing pipeline.
- xterms initiated on h1 and s1 where `tcpdump` is activated to sniff packets on interfaces h1-eth0 and s1-eth2 respectively (s1-eth2 is connected to host 2).

EXPECTED RESULTS

We expect h2 to receive an Interest just as it was emitted, with the exception of the first byte after the Ethernet header, which should have changed from 0x05, which is the assigned type for Interest packets [4], to 0xfe, inline with what we stated earlier. This fact will attest for the correctness of the parser. The fact no header was added, suppressed or shuffled demonstrates a correct deparser.

On the mininet interface, we ran the command `h1 ./rawpkt -i h1-eth0 -n "NameHere"` and instructed tcpdump to capture packets using specific flags.

- `-X` instructs tcpdump to print the contents of the packet in hexadecimal, excluding the Ethernet header.
- `-nn` instructs tcpdump not to resolve any host names or port names.
- `-i <interface>` States that tcpdump should listen only on the specified interface.
- `ether proto 0x8624` Capture only the packets whose ethertype equals 0x8624 (ethertype for NDN packets).

```
"Node: h1"
user@lasige-OptiPlex-3020:~/PEI/tutorials/P4D2_2017/
exercises/v3/build$ tcpdump -X -nn -i h1-eth0 ether proto
0x8624
tcpdump: verbose output suppressed, use -v or -vv for full
protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture
size 262144 bytes
18:35:55.739349 00:04:00:00:00:00 > 00:04:00:00:00:01,
ethertype Unknown (0x8624), length 34:
     0x0000:  0512 070a 0808 4e61 6d65 4865 7265 0a04
     0x0010:  2ced 149c
     ......NameHere..,...
```

Figure 5.2: Terminal at h1 after the packet is sent.

```
"Node: s1" (root)
user@lasige-OptiPlex-3020:~/PEI/tutorials/P4D2_2017/
exercises/v3/build$ tcpdump -X -nn -i s1-eth2 ether proto
0x8624
tcpdump: verbose output suppressed, use -v or -vv for full
protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture
size 262144 bytes
18:35:55.739883 00:04:00:00:00:00 > 00:04:00:00:00:01,
ethertype Unknown (0x8624), length 34:
     0x0000:  fe12 070a 0808 4e61 6d65 4865 7265 0a04
     0x0010:  2ced 149c
     ......NameHere..,...
```

Figure 5.3: Terminal at s1 after the packet is sent.

We verify the packet is sent out towards h2 exactly the same, but with its first byte altered to the value we predicted, thus proving our parser and deparser are functioning properly.

5.4.2 FIB

In this test, we aim to prove that our FIB operates according to what is expected.

2. FIB – Test description

SETTINGS

- Three hosts (h1, h2 and h3) are connected through s1 running FixedLengthParser, our main ingress pipeline (TopIngress) and our main deparser.
- Empty Egress pipeline.
- xterm initiated on s1 where `tcpdump` is activated to sniff packets.

- `makeFIBrules.py` transformed these entries (then added to the fib table):

  /portugal/unlisboa/fciencia/index.ht 1
  /portugal/unlisboa/fciencia 2
  /portugal/unlisboa 3

- h1 will send an Interest packet for name "/portugal/unlisboa/fciencia/index.ht".
- h1 will send an Interest for name "/portugal/unlisboa/fciencia".

EXPECTED RESULTS
If the FIB is built correctly, then a packet sent for "/portugal/unlisboa/fciencia/index.ht" will match the first entry of the file, which is the longest prefix route. The second packet should be forwarded to h2. Therefore, tcpdump should capture the packet going in and out the same interface. If this occurs, then the test also demonstrates the correctness of the control flow for an Interest seen for the first time.

In the mininet terminal, we wrote:

```
h1 ./rawpkt -i h1-eth0 -n "portugal/unlisboa/fciencia/index.ht"
```

(the -n option is actually unneeded, as the name defaults to the one we specified)

```
"Node: s1" (root)
user@lasige-OptiPlex-3020:~/PEI/tutorials/P4D2_2017/
exercises/v3/build$ tcpdump -X -nn -i s1-eth1 ether proto
0x8624
tcpdump: verbose output suppressed, use -v or -vv for full
protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture
size 262144 bytes
14:18:17.926707 00:04:00:00:00:00 > 00:04:00:00:00:01,
ethertype Unknown (0x8624), length 64:
     0x0000:  0530 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 0a04 0380
     0x0030:  bb1a
     .0.(..portugal..unlisboa..fciencia..index.ht......
14:18:17.927389 00:04:00:00:00:00 > 00:04:00:00:00:01,
ethertype Unknown (0x8624), length 64:
     0x0000:  fe30 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 0a04 0380
     0x0030:  bb1a
     .0.(..portugal..unlisboa..fciencia..index.ht......
```

Figure 5.4: Terminal at s1 in experiment FIB, after h1 sends the first packet.

The packet is received through interface s1-eth1, connected to host h1, and then for-

warded back to it. We can conclude that the longest prefix from the fib entries was selected.

After this experiment, we also made h1 send a packet towards "portugal/unlisboa/fciencia". According to our entries file, it should be sent to s1-eth2 towards h2, and, capturing packets on that interface, we verified that this is what happened. We also verified through the log that packets for unknown names would be dropped.

We can conclude that the FIB works correctly and the ingress processing flow is correct for Interests seen for the first time.

5.4.3 Egress pipeline

Our egress is relatively simple. It only applies tables to set source MAC and destination MAC. This is a typical task on most P4 programs, so the test is trivial. Because the switch's MAC varies, we do not apply a source MAC table, but table dmac demonstrates the correct processing just the same.

With only a single host and a switch, and using the FIB entries from experiment 2, we made h1 send a single packet towards the switch.

```
h1 ./rawpkt -i h1-eth0 -d ?
```

Our packet generator program does some math and deterministically maps '?' to a fixed hex value. Remember the Interest is being sent out to its default name and, according to our FIB entries file, the switch FIB determines that the packet flows back to h1. When received, we see the packet's destination MAC equals h1's, hence upholding the correct operation of the egress.

5.4.4 Ingress pipeline

In this test, we aim to prove, as a whole, the correct main processing flow for packets. Namely, spurious Data should be eliminated, repeated Interests should not be forwarded, but update the PIT, and Data should be cached correctly.

4. PIT and CS

SETTINGS

- Two hosts (h1, h2) are interconnected through switch s1, which runs our program.
- Same FIB entries as in experiment 2.
- A file called `data` exists in the test directory with 8 bytes of content: "HaveData".
- Two xterms initiated on s1 where `tcpdump` is activated to sniff packets, one on interface s1-eth1, the other on interface s1-eth2.

PROCEDURE

1. h1 will send an unwarranted Data packet.
2. h2 will send an Interest for the default name.
3. h2 will send another Interest for the default name.

4. h1 will reply with Data for the default name.
5. h2 will send yet another Interest for the default name.

EXPECTED RESULTS
We are aiming to prove several things here.

PHASE I – SPURIOUS DATA: h1 sent an unwarranted Data. We expect the switch to drop it.
PHASE II – REPEATED INTEREST: On step 3, h2 is sending a repeated Interest. By consulting the PIT, the switch knows this interface is already signaled, therefore it should not forward the packet.
PHASE III – REQUEST-RESPONSE: Steps 2 and 4 are the primary use case for the processing pipeline: some host sends an Interest and some other host replies with Data. h2 emitted the Interest and should receive Data accordingly.
PHASE IV – CACHE: Step 5 tests the ability of our emulated NDN router to cache Data content. Instead of forwarding h2's Interest anew, the switch should send its cached Data.

With our test environment set up as specified above, we wrote the following in the Mininet terminal:

```
h1 ./rawpkt -i h1-eth0 -f data -d 4:0:0:0:1
h2 ./rawpkt -i h2-eth0 -d 4:0:0:0:0
h2 ./rawpkt -i h2-eth0 -d 4:0:0:0:0 (intentional repetition)
h1 ./rawpkt -i h1-eth0 -f data -d 4:0:0:0:1
h2 ./rawpkt -i h2-eth0 -d -d 4:0:0:0:1
```

We verified that the spurious Data was successfully dropped by inspecting the switch log, and noting the following lines:

```
[16:38:55.041] [bmv2] [T] [thread 8676] [60.0] [cxt 0] Ingress.p4(269)
Condition "currentPortList == 0 || storedHash != m.hashed_components" is
true
[16:38:55.041] [bmv2] [T] [thread 8676] [60.0] [cxt 0] Applying table
'tbl_Drop'
[16:38:55.041] [bmv2] [D] [thread 8676] [60.0] [cxt 0] Dropping packet at
the end of ingress
```

The switch inspected the PIT register contents and placed them in `currentPortList`. Verifying it was zero, it executed an explicit call to *Drop*(). This proves **PHASE I** ran without errors and the processing flow for spurious Data is correct.

In appendix B.1 we can see eight records that demonstrate all phases achieve correct results. The captured instances refer to the following events.

1. h1's spurious Data is received.

2. h2's Interest is received.
3. h2's Interest is forwarded through interface s1-eth1. (Notice the nonce is the same.)
4. h2's repeated Interest is received.
5. h1's Data is received.
6. h1's Data is sent towards h2.
7. h2's Interest is received.
8. A cached Data is sent towards h2.

Notice, first, that the switch did not forward the spurious Data in step 1. Second, the switch did not send out a response in 5. Inspecting the log again, we find the following line:

```
[17:33:22.800] [bmv2] [T] [thread 9550] [61.0] [cxt 0] Ingress.p4(244)
Condition "currentPortList == 0" is false
```

Finding the current bit array was not zero, the program interpreted as already having forwarded the Interest upstream. In this case, it should not forward again. Thus demonstrating that the switch processes repeated Interests correctly.

The last instance shown by tcpdump also proves caching is working. Together, all this shows the processing pipeline for Interests and Data, using registers to define the PIT and the Content Store, has been implemented correctly.

5.4.5 Multicast and Content Store in the SimpleSwitch

To implement the CS as a C++ class, we edited the simple_switch code directly. We also implement multicasting of the Data packet. As mentioned in chapter 4, we had to sacrifice discarding Interest packets desiring the same name, so the router always sweeps Interest packets through the FIB and forwards them upstream.

The tests for multicast are trivial, so they are mingled with the content store tests.

5. Multicast and caching

SETTINGS

- Four hosts (h1–h4) are connected through the modified switch s1 running our P4 program.
- xterm initiated on h1, h2, h3 and h4 where `tcpdump` is activated to sniff packets.
- FIB entries are the same as the former experience.

PROCEDURE

- h2 and h3 will send Interests for the default name ("portugal/unlisboa/fciencia/index.ht").
- h1 will send Data for the default name.
- h2 will send an Interest for "portugal/unlisboa".
- h3 will reply with Data for that name.
- h2 will repeat the requests for the default name and "portugal/unlisboa".

EXPECTED RESULTS
Recall the FIB is forwarding the default name to h1 and "portugal/unlisboa" to h3. In step 2, we expect the Data packet to be multicast to hosts 2 and 3, but not h4 or h1.
Then, on the last two steps, the router should reply with Data packets directly without forwarding them to h1 or h3.

On the Mininet terminal, we wrote:

```
h2 ./rawpkt -i h2-eth0
h3 ./rawpkt -i h3-eth0
h1 ./rawpkt -i h1-eth0 -f mega
h2 ./rawpkt -i h2-eth0 -n "portugal/unlisboa"
h3 ./rawpkt -i h3-eth0 -f data -n "portugal/unlisboa"
h2 ./rawpkt -i h2-eth0
h2 ./rawpkt -i h2-eth0 -n "portugal/unlisboa"
```

(*mega* is a file with the string "Alfomega" written within.)

The results of this experiment, as presented in a consolidated log under appendix B.2, match our predictions.

In a first interaction, host h2 sends (1.1) an Interest that is received (1.2) by h1. Host h3 also sends (2.1) an Interest which is also received (2.2) by h1. As discussed, in a real implementation, this shouldn't happen (and in our former experiments, it doesn't). The router should have detected it had already sent out an Interest for the default name and drop the packet from h3. This is a necessary consequence of being unable to implement the extern calls.

In a second interaction, h1 sends (3.1) Data. This Data is correctly received by h2 (3.2) and h3 (3.3). No records appear for h4. Therefore, our multicast worked properly.

Following this, h2 will send an Interest for name "portugal/unlisboa" (4.1) and it is forwarded to h3, who receives it (4.2) accordingly. h3 replies with Data (5.1) and it is received by h2 (5.2).

Now, h2 starts repeating requests. All the records we labeled as 6 were captured on h2's interface. The log shows h2 sending an Interest for the default name (6.1) and the router replying back with the respective Data (6.2). Notice the string is "AlfoMega" in the alphanumeric print, which, if you notice the procedure and the above commands, correctly corresponds to the piece of information we associated with that name.

Then, h2 requests "portugal/unlisboa" (6.3). Again, the router finds it has cached this Data, so it replies (6.4) with the Data in store. This demonstrates the correct operation of content archiving and retrieval.

5.5 Summary

In this section, we overviewed our testing tools (§5.1) and contextualized their role; described the environment used to run the tests (§5.2), and, most importantly, we evaluated the memory requirements of our solution (§5.3) and attested the correctness of all the features proposed and implemented (§5.4).

Chapter 6

Conclusion & Future Work

Named data networks provide a new communication model that is suitable for most use cases of today's Internet, such as e-commerce, digital media, social networking and smartphone apps. By routing content based on name, and promoting in-network caching, it radically changes the paradigm. Such change has numerous advantages, but deployment is an issue. The recent emergence of programmble switches gives the opportunity to make NDN practical. In this thesis we propose an NDN router using programmable switches. Our prototype, implemented in P4, a programming language for these switches, improves over previous work by including crucial NDN functionality previously unexplored.

Working with P4 made us understand the complexity behind modifying the switch core and adding new functionality and constructs to the language. On the other hand, this work also demonstrates that, even under all the constraints of the compilers and the software switch, it is still possible to run an NDN router that entails its main functionality. Despite the limitations of available testing tools, our evaluation has shown that the basic requirements of an NDN router were met with our implementation.

In the future we plan to explore other switch targets, namely in hardware, to further understand the potential advantages and limitations of our design. In particular, we plan to implement specific functionality, such as the Content Store, as a function in a NetFPGA, made available as an extern interface to our P4 program.

If named data networks ever become a reality, we also foresee the inclusion of more advanced features; for example, a proactive content retrieval module that takes the initiative to send Interests based on some observed time pattern. Say, if the router knows its consumers tend to access a specific web forum around 18 o'clock, then it could proactively request the latest content. This would greatly leverage the router's caching capabilities, as well as ensuring the benefits of using NDN are also felt by the end users, and not just by distributed system designers.

Bibliography

[1] BWorld Robot Control Software. https://github.com/p4lang/behavioral-model. Accessed: 2016-12-01.

[2] P4 adoption continues to grow rapidly. http://p4.org/technical-steering-committee/p4-adoption-continues-to-grow-rapidly/. Accessed: 2010-07-14.

[3] P4 Specification. http://p4.org/wp-content/uploads/2016/11/p4-spec-latest.pdf. Accessed: 2016-12-06 (version 1.0.3).

[4] Ndn project team, ndn packet format specication (version 0.1). http://named-data.net/doc/ndn-tlv/, 2014. Accessed: 2016-12-01.

[5] P4 16 language specification. http://p4.org/wp-content/uploads/2016/12/P4_16-prerelease-Dec_16.html, December 2016. Accessed: 2017-04-05.

[6] bmv2 backend does not correctly cast bool to bit<1> in if statement. https://github.com/p4lang/p4c/issues/750, 2017.

[7] Investigate why casting bit<1> fields to bool breaks stf tests. https://github.com/p4lang/p4c/issues/737, 2017.

[8] P4 16 language specification. http://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.html, May 2017. Accessed: 2017-06-13.

[9] [p4-dev] difference among p4c, p4c-bm2-ss, p4c-bmv2 ?? http://lists.p4.org/pipermail/p4-dev_lists.p4.org/2017-June/001114.html, June 2017. Accessed: 2017-06-19.

[10] [p4-dev] varbit. http://lists.p4.org/pipermail/p4-dev_lists.p4.org/2017-June/001072.html, June 2017. Accessed: 2017-06-19.

[11] [p4-dev](p4-16) about header union behavior. http://lists.p4.org/pipermail/p4-dev_lists.p4.org/2017-April/000934.html, April 2017. Accessed: 2017-05-16.

[12] p4lang/p4c. https://github.com/p4lang/p4c, 2017.

[13] Small quality of life enhancement proposals. https://github.com/p4lang/p4c/issues/585, 2017.

[14] v1model hash() yielding frontend compiler bug. https://github.com/p4lang/p4c/issues/584, 2017.

[15] v1model user-defined metadata as subparser invocation parameter produces compiler bug (bmv2 backend). https://github.com/p4lang/p4c/issues/698, 2017.

[16] Alexander Afanasyev, Junxiao Shi, Beichuan Zhang, Lixia Zhang, Ilya Moiseenko, Yingdi Yu, Wentao Shang, Yi Huang, Jerald Paul Abraham, Steve DiBenedetto, et al. Nfd developer's guide. Technical report, Technical Report NDN-0021, NDN, 2014.

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[18] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 99–110. ACM, 2013.

[19] Michael J Demmer, Kevin R Fall, Teemu Koponen, and Scott Shenker. Towards a modern communications api. In *HotNets*. Citeseer, 2007.

[20] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM Sigplan Notices*, volume 46, pages 279–291. ACM, 2011.

[21] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2009.

[22] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[23] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[24] Aurojit Panda, Colin Scott, Ali Ghodsi, Teemu Koponen, and Scott Shenker. Cap for networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 91–96. ACM, 2013.

[25] Fernando M.V. Ramos. User-centric programmable virtual networks. Retrieved from http://www.navigators.di.fc.ul.pt/w2/img_auth.php/2/26/2017.06.14_uPVN_Navtalk.pdf, 2017.

[26] Salvatore Signorello, Jerome Francois, Olivier Festor, et al. Ndn. p4: Programming information-centric data-planes. In *International Workshop on Open-Source Software Networking (OSSN), IEEE International Conference on Network Softwarization*, 2016.

[27] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. Smoking out the heavy-hitter flows with hashpipe. *arXiv preprint arXiv:1611.04825*, 2016.

[28] Mininet Team. Mininet: An instant virtual network on your laptop (or other pc), 2012.

[29] Matteo Varvello, Diego Perino, and Jairo Esteban. Caesar: A content router for high speed forwarding. In *Proceedings of the Second Edition of the ICN Workshop on Information-centric Networking*, ICN '12, pages 73–78, New York, NY, USA, 2012. ACM.

[30] Cheng Yi, Alexander Afanasyev, Lan Wang, Beichuan Zhang, and Lixia Zhang. Adaptive forwarding in named data networking. *ACM SIGCOMM computer communication review*, 42(3):62–67, 2012.

[31] Haowei Yuan and Patrick Crowley. Scalable pending interest table design: From principles to practice. In *INFOCOM, 2014 Proceedings IEEE*, pages 2049–2057. IEEE, 2014.

[32] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, Patrick Crowley, Christos Papadopoulos, Lan Wang, Beichuan Zhang, et al. Named data networking. *ACM SIGCOMM Computer Communication Review*, 44(3):66–73, 2014.

Appendix A

A.1 Our main parser as it parses name components

```
parser TopParser(packet_in b, inout Metadata m, inout
    standard_metadata_t stdm, out Parsed_packet p) {

    Subparser TLVreader(); //Instantiation
    bit<HASH_LENGTH> hashOutput; //Attribute

    ...

    //m.num_of_components was initialized to 0.
    state name_parsing {
      verify(m.num_of_components < MAX_COMPONENTS, error.
          NumberOfComponentsAboveMaximum);

      TLVreader.apply(b, m, p.components[i]);

      //Use the value of the header union member that is active
      transition select(m.last_extracted_TLV_lencode) {
        253 : hash_medium_TLV;
        254 : hash_large_TLV;
        //The subparser terminated processing if 255
        _   : hash_small_TLV;
      }
    }

    state hash_small_TLV {
      hash(hashOutput, HashAlgorithm.crc32, 1, p.components[m.
          num_of_components].smallTLV.value, 0xffFFffFF);
      transition post_hash;
    }

    state hash_medium_TLV {
      hash(hashOutput, HashAlgorithm.crc32, 1, p.components[m.
          num_of_components].mediumTLV.value, 0xffFFffFF);
      transition post_hash;
    }
```

```
    state hash_small_TLV {
      hash(hashOutput, HashAlgorithm.crc32, 1, p.components[m.
          num_of_components].largeTLV.value, 0xffFFffFF);
      transition post_hash;
    }

    state post_hash {
      m.namesize = m.namesize - m.last_extracted_TLV_size - m.
          last_extracted_TLV_extension - 2;

      m.hashed_components = m.hashed_components << HASH_LENGTH;
      m.hashed_components = m.hashed_components | hashOutput;

      m.num_of_components = m.num_of_components + 1;

      transition select(m.namesize) {
        0 : parse_ndn_pkt_by_type;
        _ : name_parsing;
      }
    }

    state parse_ndn_pkt_by_type {
      m.hashed_components = m.hashed_components << ((
          MAX_COMPONENTS - m.num_of_components) * HASH_LENGTH);

      ...
    }
}
```

Appendix B

B.1 Merged tcpdump logs sniffing on switch interfaces s1-eth1 and s1-eth2 in test 4.

```
"Node:  s1" (root)
17:33:17.734966 00:04:00:00:00:00 > 00:04:00:00:00:01,
ethertype Unknown (0x8624), length 68:
     0x0000:  0634 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 1508 4861
     0x0030:  7665 4461 7461 bb1a
     .4.(..portugal..unlisboa..fciencia..index.ht..HaveData
17:33:21.211574 00:04:00:00:00:01 > 00:04:00:00:00:00,
ethertype Unknown (0x8624), length 64:
     0x0000:  0530 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 0a04 04fe
     0x0030:  3bc8
     .0.(..portugal..unlisboa..fciencia..index.ht....;.
17:33:21.214215 00:04:00:00:00:01 > 00:04:00:00:00:00,
ethertype Unknown (0x8624), length 64:
     0x0000:  0530 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 0a04 04fe
     0x0030:  3bc8
     .0.(..portugal..unlisboa..fciencia..index.ht....;.
17:33:22.799834 00:04:00:00:00:01 > 00:04:00:00:00:00,
ethertype Unknown (0x8624), length 64:
     0x0000:  0530 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 0a04 326d
     0x0030:  7b63
     .0.(..portugal..unlisboa..fciencia..index.ht..2m{c
17:33:24.840607 00:04:00:00:00:00 > 00:01:00:00:00:01,
ethertype Unknown (0x8624), length 68:
     0x0000:  0634 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
```

```
        0x0020:  6961 0808 696e 6465 782e 6874 1508 4861
        0x0030:  7665 4461 7461
        .4.(..portugal..unlisboa..fciencia..index.ht..HaveData
17:33:24.841958 00:04:00:00:00:00 > 00:04:00:00:00:01,
ethertype Unknown (0x8624), length 68:
        0x0000:  0634 0728 0808 706f 7274 7567 616c 0808
        0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
        0x0020:  6961 0808 696e 6465 782e 6874 1508 4861
        0x0030:  7665 4461 7461
        .4.(..portugal..unlisboa..fciencia..index.ht..HaveData
17:33:27.176590 00:04:00:00:00:01 > 00:04:00:00:00:00,
ethertype Unknown (0x8624), length 64:
        0x0000:  0530 0728 0808 706f 7274 7567 616c 0808
        0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
        0x0020:  6961 0808 696e 6465 782e 6874 0a04 57e6
        0x0030:  6dd7
        .0.(..portugal..unlisboa..fciencia..index.ht..W.m.
17:33:27.177828 00:04:00:00:00:01 > 00:04:00:00:00:01,
ethertype Unknown (0x8624), length 68:
        0x0000:  0534 0728 0808 706f 7274 7567 616c 0808
        0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
        0x0020:  6961 0808 696e 6465 782e 6874 1508 4861
        0x0030:  7665 4461 7461
        .4.(..portugal..unlisboa..fciencia..index.ht..HaveData
```

B.2 tcpdump logs sniffing on interfaces h1-eth0, h2-eth0 and h3-eth0 in test 5 (the labels in square brackets and question marks are artificial), merged by increasing timestamp.

```
"Node:  s1" (root)
[1.1] 15:35:56.995873 00:04:00:00:00:01 > ?, ethertype
Unknown (0x8624), length 64:
        0x0000:  0530 0728 0808 706f 7274 7567 616c 0808
        0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
        0x0020:  6961 0808 696e 6465 782e 6874 0a04 2dc8
        0x0030:  0311
        .0.(..portugal..unlisboa..fciencia..index.ht..-...
```

```
[1.2] 15:35:56.998512 00:04:00:00:00:01 > 00:04:00:00:00:00,
ethertype Unknown (0x8624), length 64:
     0x0000:  0530 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 0a04 2dc8
     0x0030:  0311
     .0.(..portugal..unlisboa..fciencia..index.ht..-...
[2.1] 15:36:04.115695 00:04:00:00:00:02 > ?, ethertype
Unknown (0x8624), length 64:
     0x0000:  0530 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 0a04 5d5d
     0x0030:  baad
     .0.(..portugal..unlisboa..fciencia..index.ht..]]..
[2.2] 15:36:04.118223 00:04:00:00:00:02 > 00:04:00:00:00:00,
ethertype Unknown (0x8624), length 64:
     0x0000:  0530 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 0a04 5d5d
     0x0030:  baad
     .0.(..portugal..unlisboa..fciencia..index.ht..]]..
[3.1] 15:36:10.363525 00:04:00:00:00:00 > ?, ethertype
Unknown (0x8624), length 68:
     0x0000:  0634 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 1508 416c
     0x0030:  666f 4d65 6761
     .4.(..portugal..unlisboa..fciencia..index.ht..AlfoMega
[3.2] 15:36:10.366148 00:04:00:00:00:00 > 00:04:00:00:00:01,
ethertype Unknown (0x8624), length 68:
     0x0000:  0634 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 1508 416c
     0x0030:  666f 4d65 6761
     .4.(..portugal..unlisboa..fciencia..index.ht..AlfoMega
[3.3] 15:36:10.366668 00:04:00:00:00:00 > 00:04:00:00:00:02,
ethertype Unknown (0x8624), length 68:
     0x0000:  0634 0728 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 1508 416c
     0x0030:  666f 4d65 6761
     .4.(..portugal..unlisboa..fciencia..index.ht..AlfoMega
[4.1] 15:36:20.619959 00:04:00:00:00:01 > ?, ethertype
Unknown (0x8624), length 44:
     0x0000:  051c 0714 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0a04 7b62 a902
     ......portugal..unlisboa..{b..
[4.2] 15:36:20.622430 00:04:00:00:00:01 > 00:04:00:00:00:02,
```

```
ethertype Unknown (0x8624), length 44:
     0x0000:  051c 0714 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0a04 7b62 a902
     ......portugal..unlisboa..{b..
[5.1] 15:36:30.029521 00:04:00:00:00:02 > 00:04:00:00:00:01,
ethertype Unknown (0x8624), length 48:
     0x0000:  0620 0714 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 1508 4861 7665 4461
     0x0020:  7461
     ......portugal..unlisboa..HaveData
[5.2] 15:36:30.027485 00:04:00:00:00:02 > ?, ethertype
Unknown (0x8624), length 48:
     0x0000:  0620 0714 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 1508 4861 7665 4461
     0x0020:  7461
     ......portugal..unlisboa..HaveData
[6.1] 15:36:33.920016 00:04:00:00:00:01 > ?, ethertype
Unknown (0x8624), length 64:
     0x0000:  0620 0714 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 0a04 4ef5
     0x0030:  86f3
     .0.(..portugal..unlisboa..fciencia..index.ht..N...
[6.2] 15:36:33.920681 ?  > 00:04:00:00:00:01, ethertype
Unknown (0x8624), length 68:
     0x0000:  0620 0714 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0808 6663 6965 6e63
     0x0020:  6961 0808 696e 6465 782e 6874 1508 416c
     0x0030:  666f 4d65 6761
     .4.(..portugal..unlisboa..fciencia..index.ht..AlfoMega
[6.3] 15:36:36.507880 00:04:00:00:00:01 > ?, ethertype
Unknown (0x8624), length 44:
     0x0000:  051c 0714 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 0a04 1908 47ea
     ......portugal..unlisboa....G.
[6.4] 15:36:36.510124 ?  > 00:04:00:00:00:01, ethertype
Unknown (0x8624), length 48:
     0x0000:  0620 0714 0808 706f 7274 7567 616c 0808
     0x0010:  756e 6c69 7362 6f61 1508 4861 7665 4461
     0x0020:  7461
     ......portugal..unlisboa..HaveData
```